Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

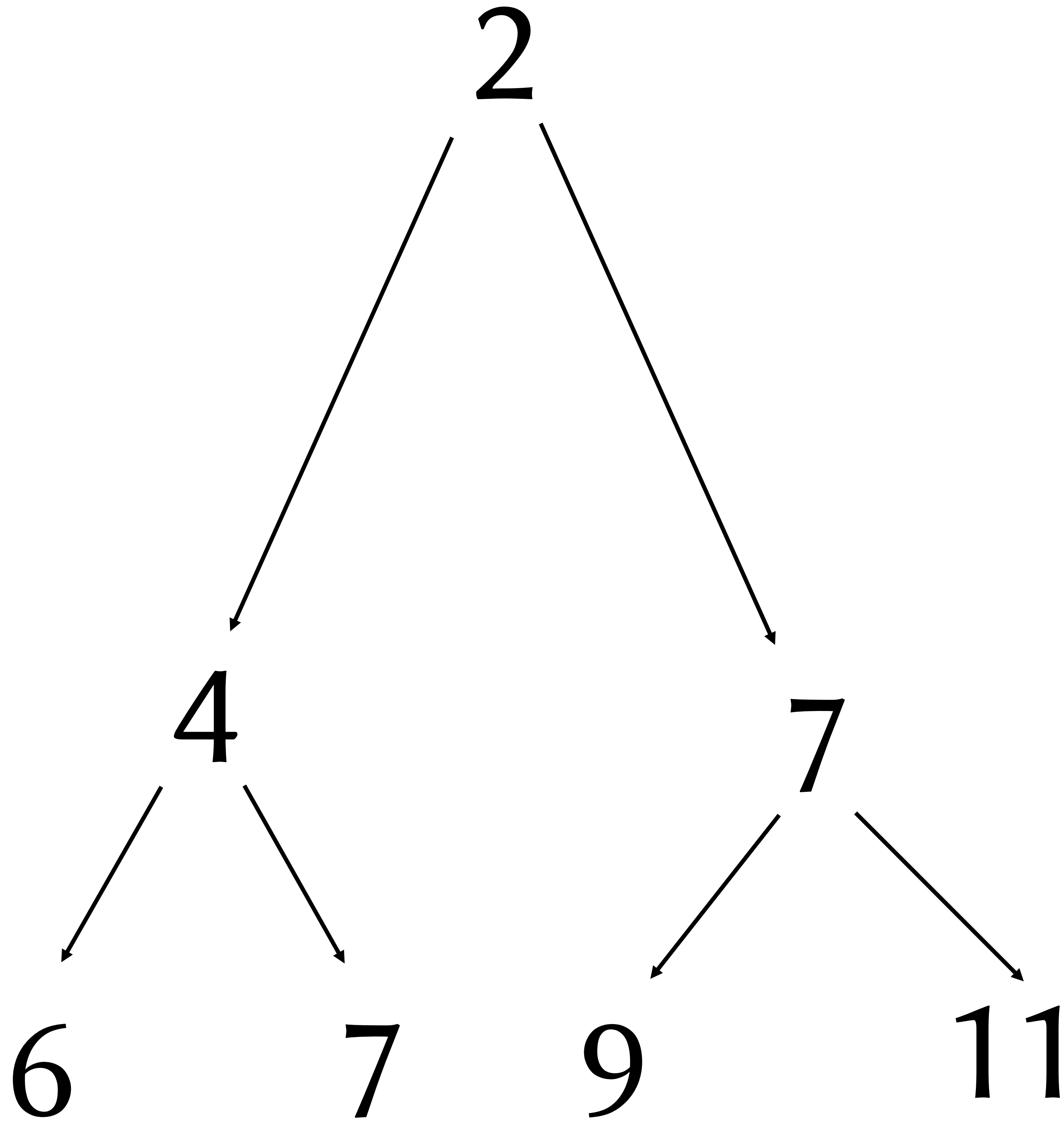
November 2, 2017

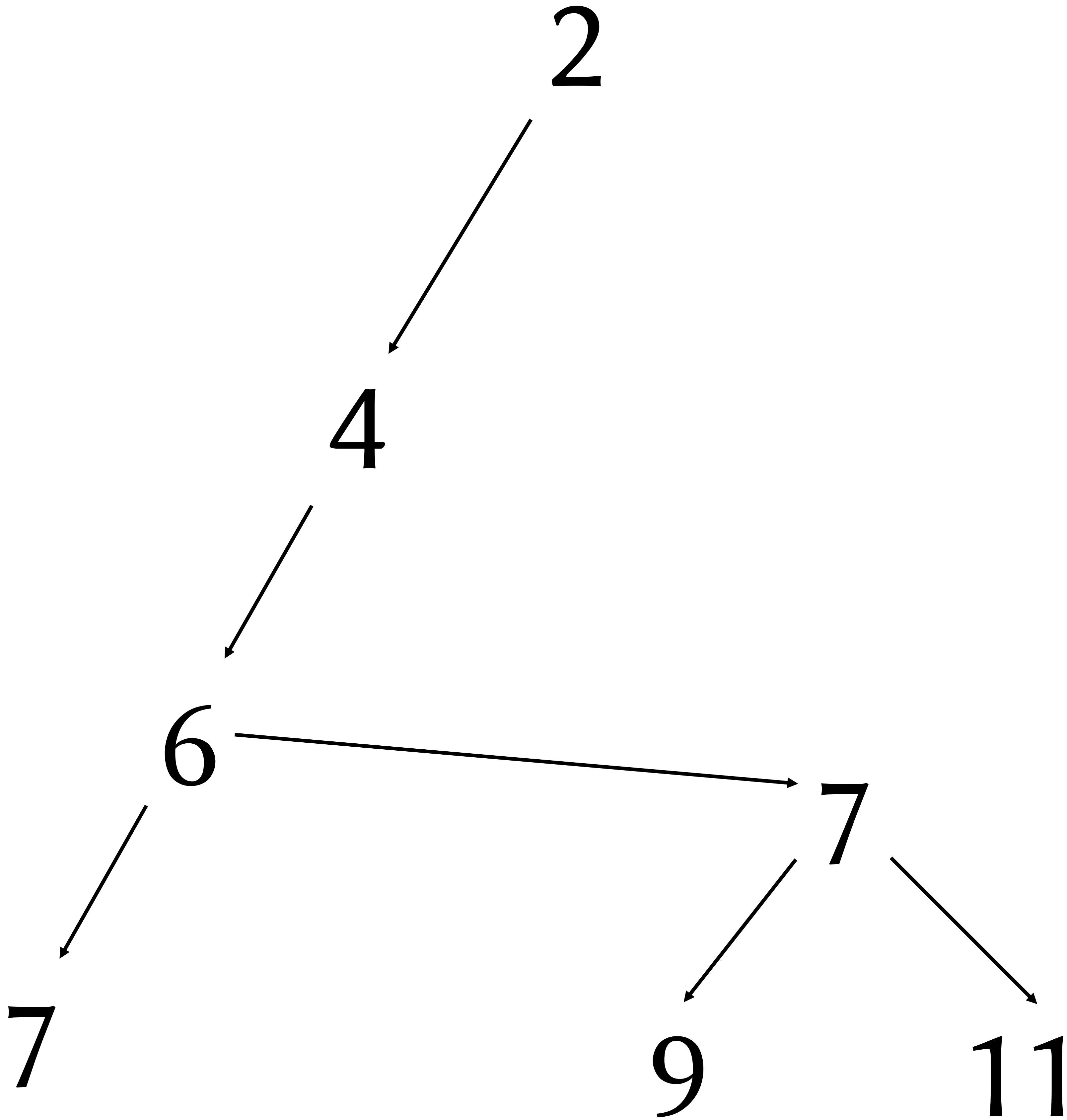
Functional Data Structures

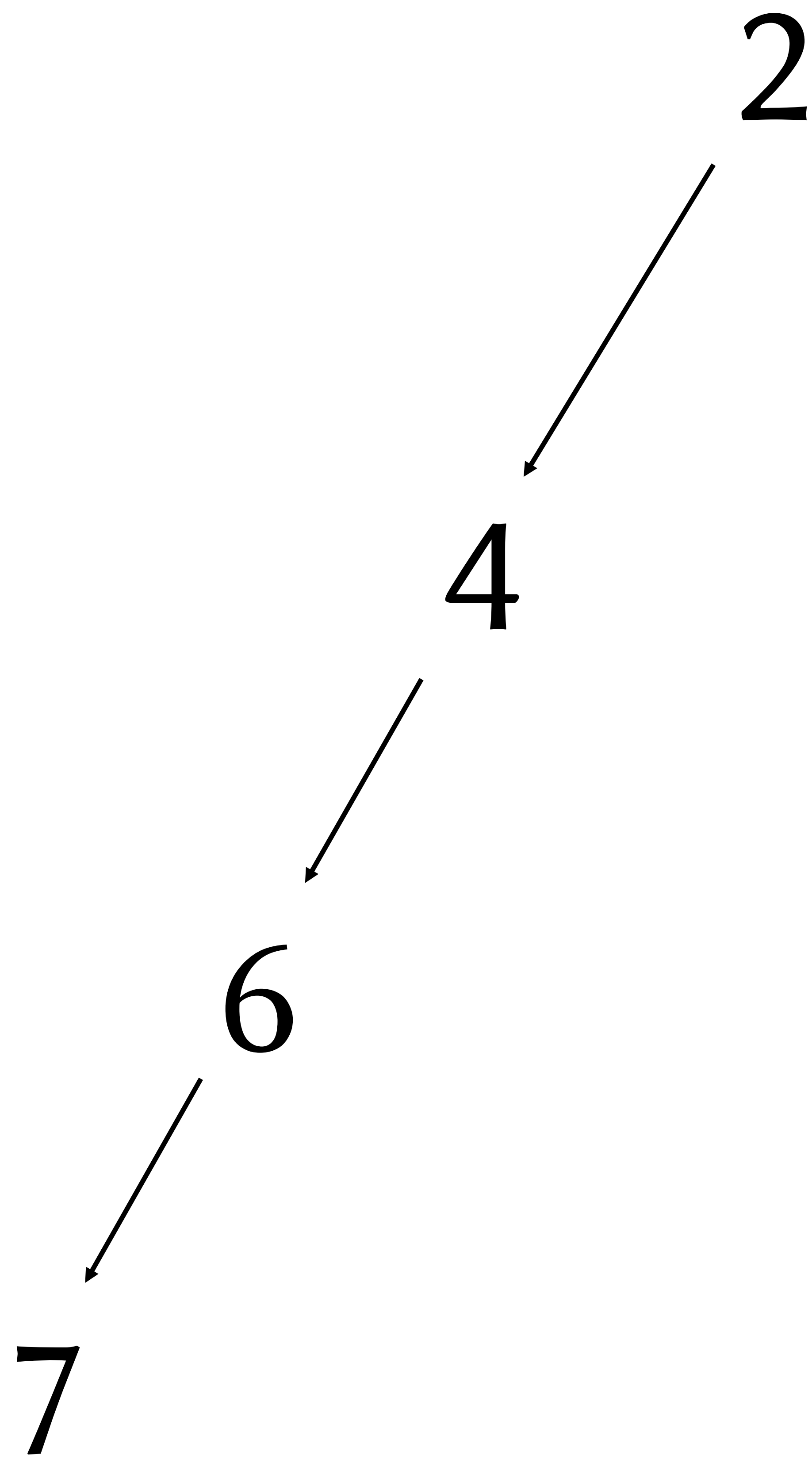
Leftist Heaps

Leftist Heaps

- Often in a collection of elements we only need to access the *minimum* element
- A data structure that supports access only to the minimum element is called a *heap:*
  - A tree in which the element at the root of each subtree is the minimum element of that subtree
  - *Priority queues* are often implemented as heaps

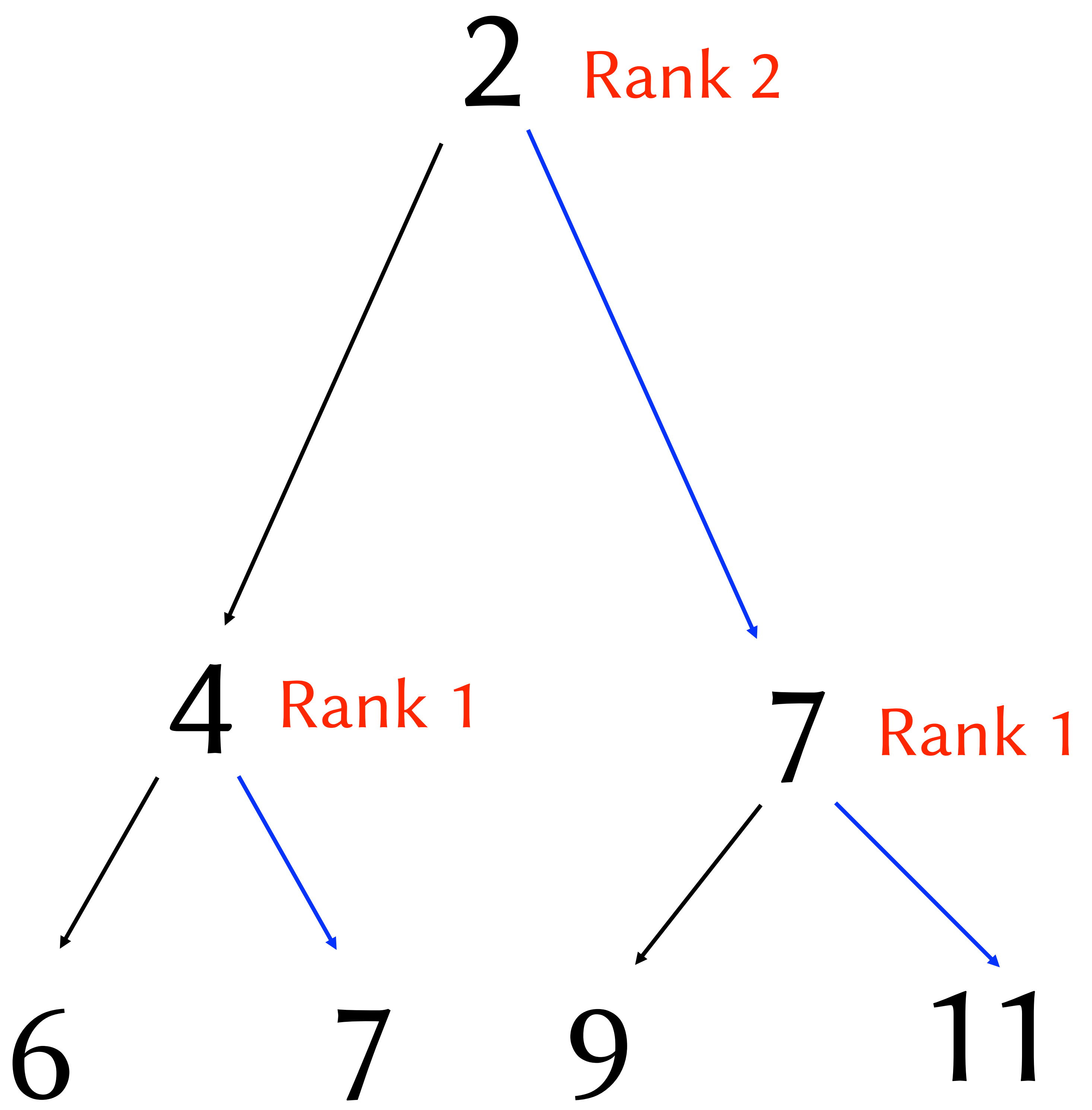

2
4
7
6
7
9
11

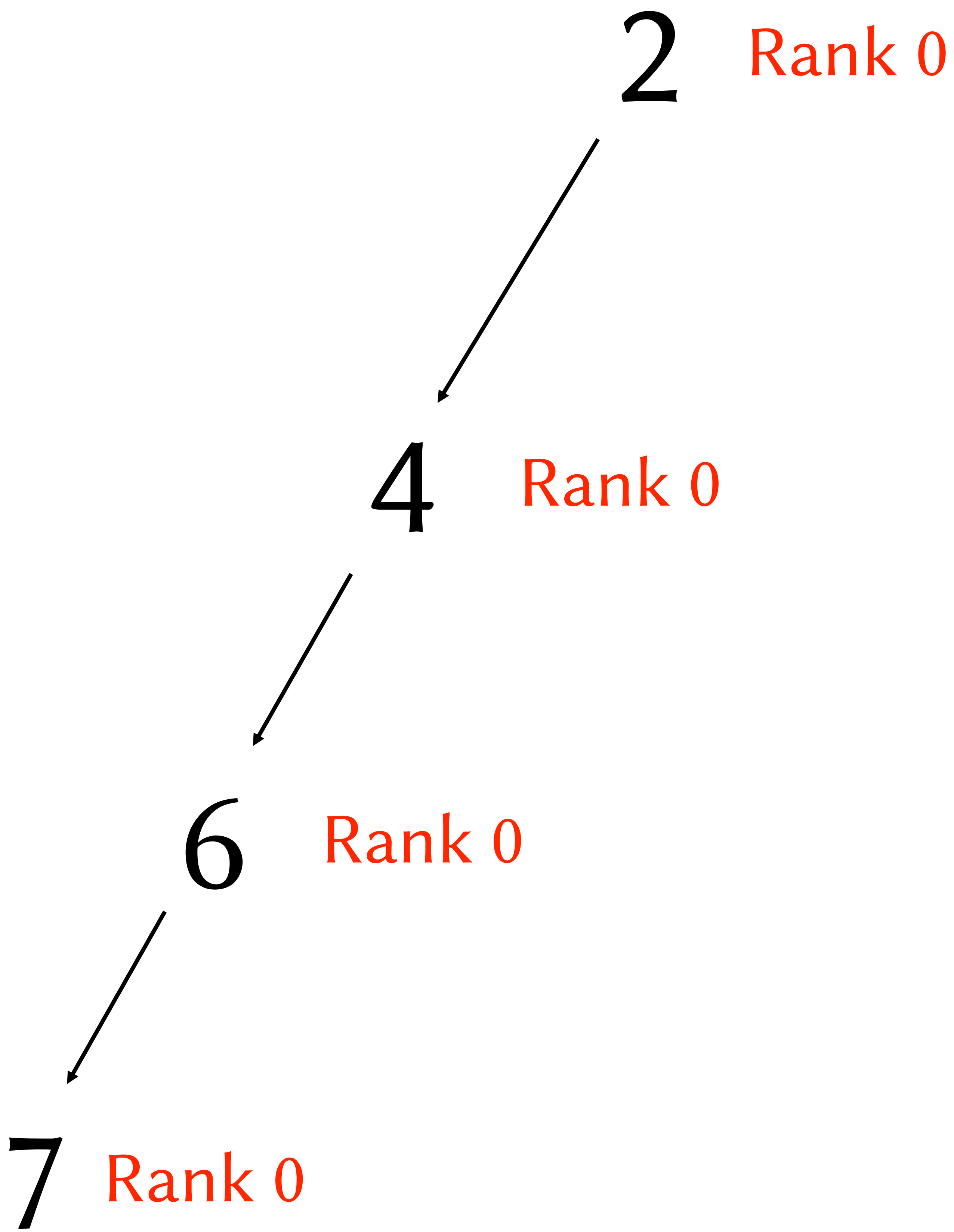

2
4
6
7
7
9
11

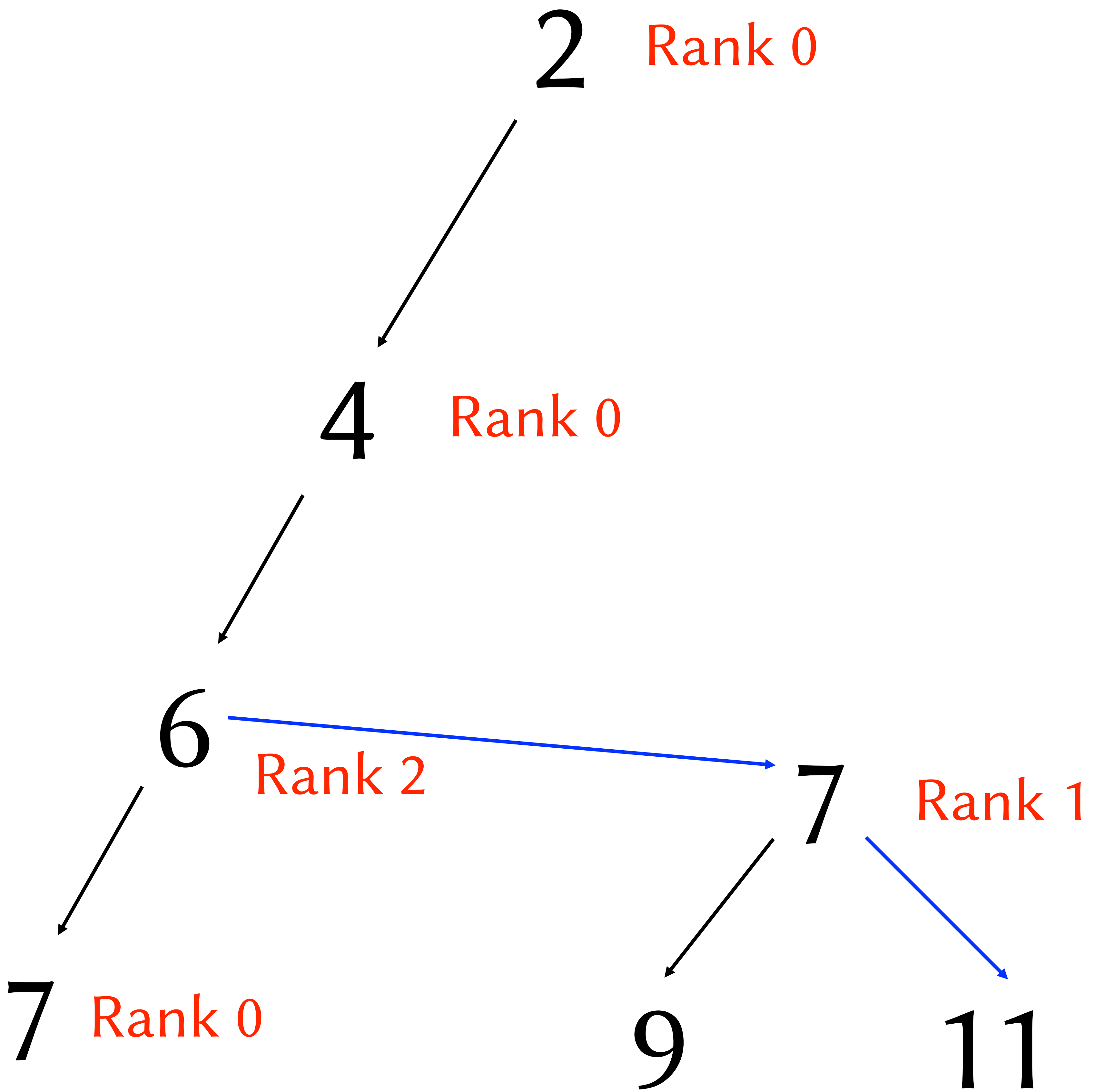

2
4
6
7

Leftist Heaps

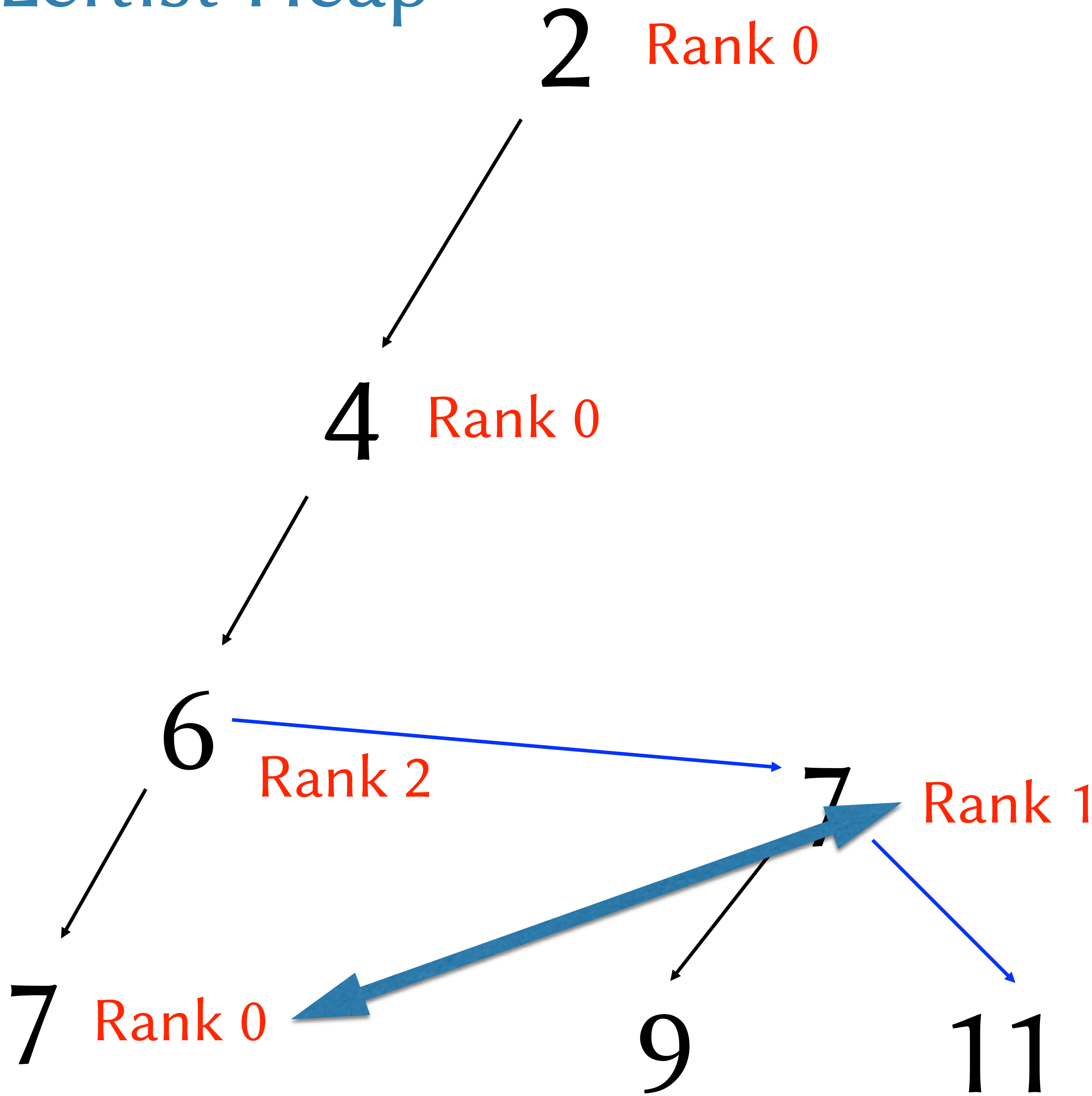
- Let the *rank* of a node be the length of its *right spine*
- Then a *leftist heap* also satisfies the following property:
  - The rank of a left child is no smaller than the rank of its sibling


2
Rank 2
4
Rank 1
7
Rank 1
6
7
9
11


2
Rank 0
4
Rank 0
6
Rank 0
7
Rank 0


2
Rank 0
4
Rank 0
6
Rank 2
7
Rank 1
7
Rank 0
9
11

Not a Leftist Heap

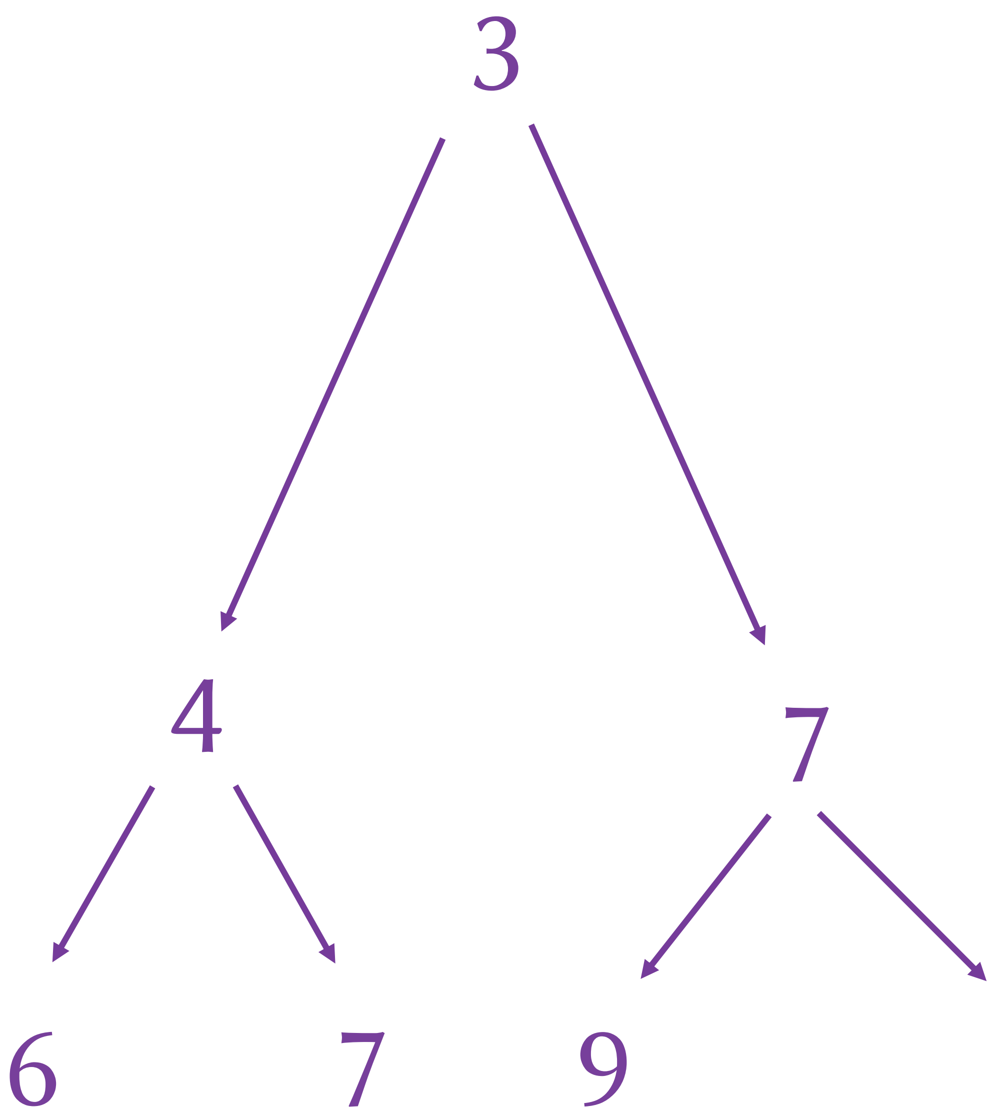


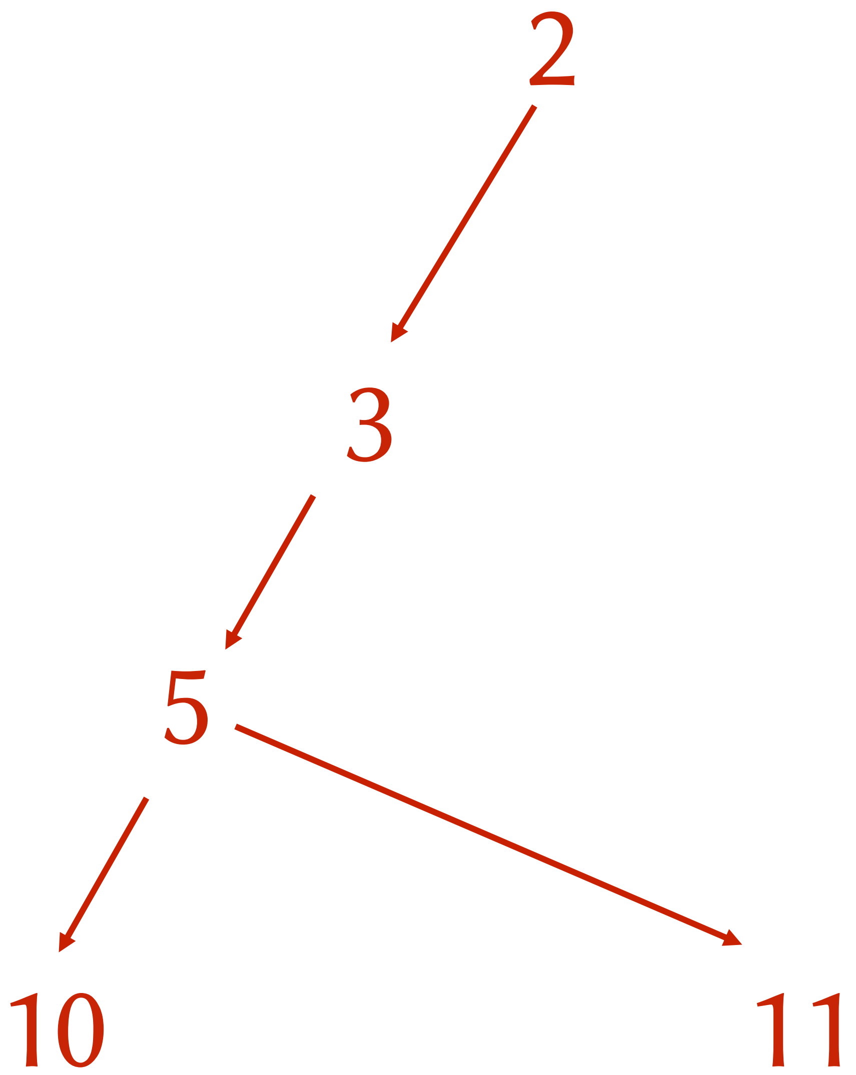
Consequences of the Leftist Property

- The right spine of a node is always the shortest path to a leaf
- The right spine of a node contains O(log n) elements in the worst case
- The elements along the right spines are in sorted order

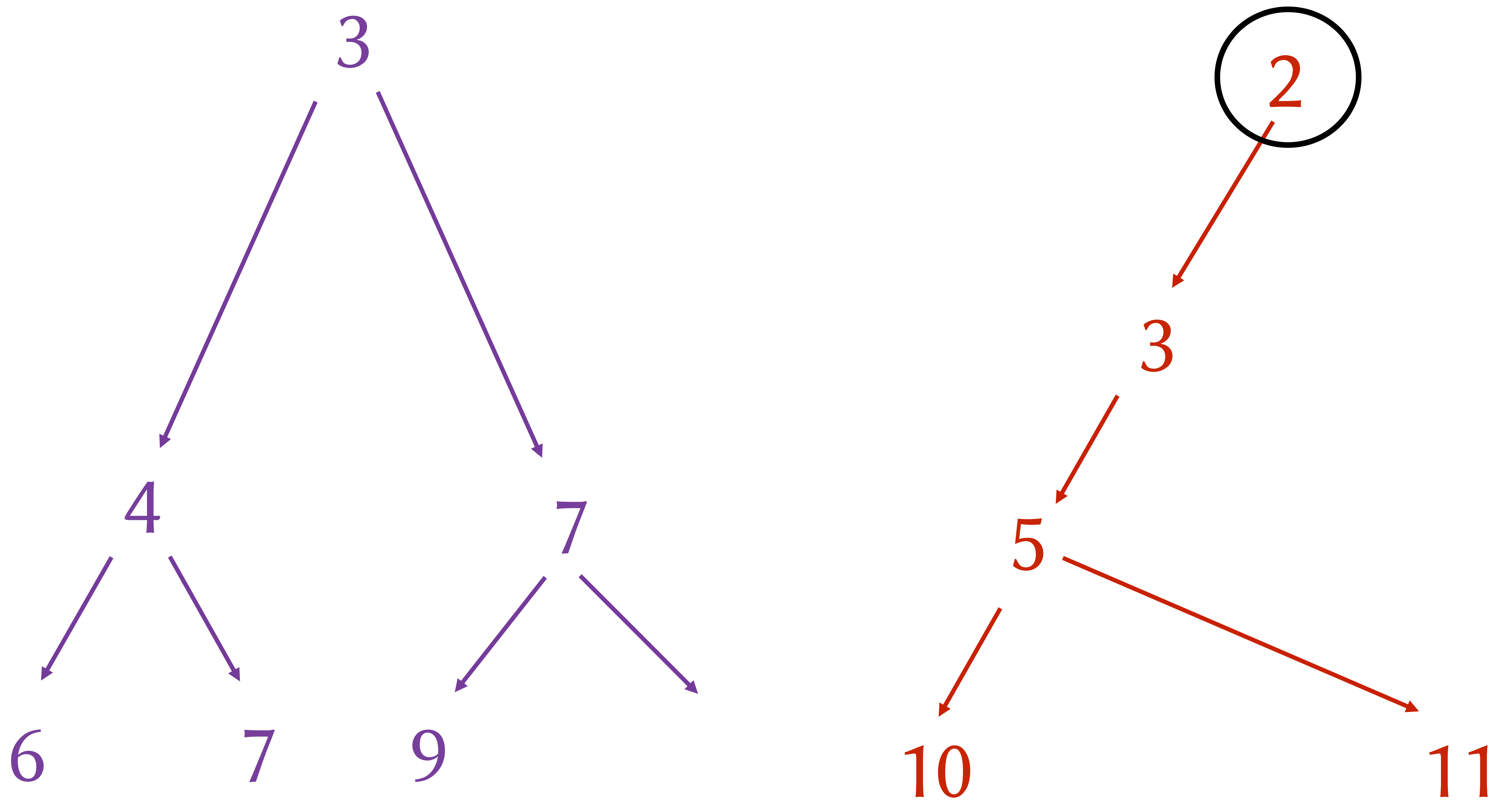
Efficient Merging of Two Leftist Heaps

- Intuitively, we can merge two leftist heaps by:
  - Merging their right spines as if they were sorted lists
  - Swapping child nodes along the merged right spine as needed to preserve the leftist property

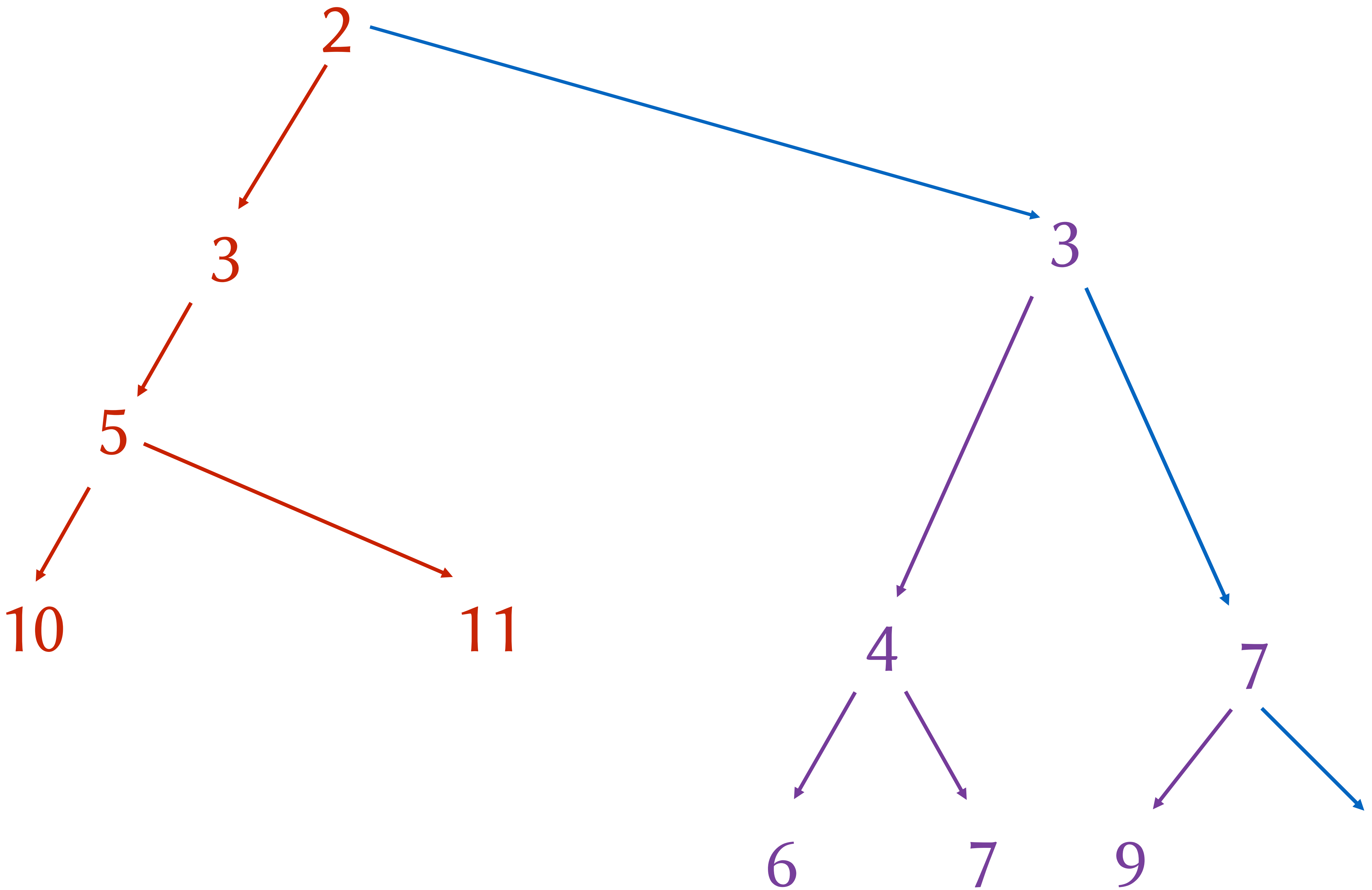




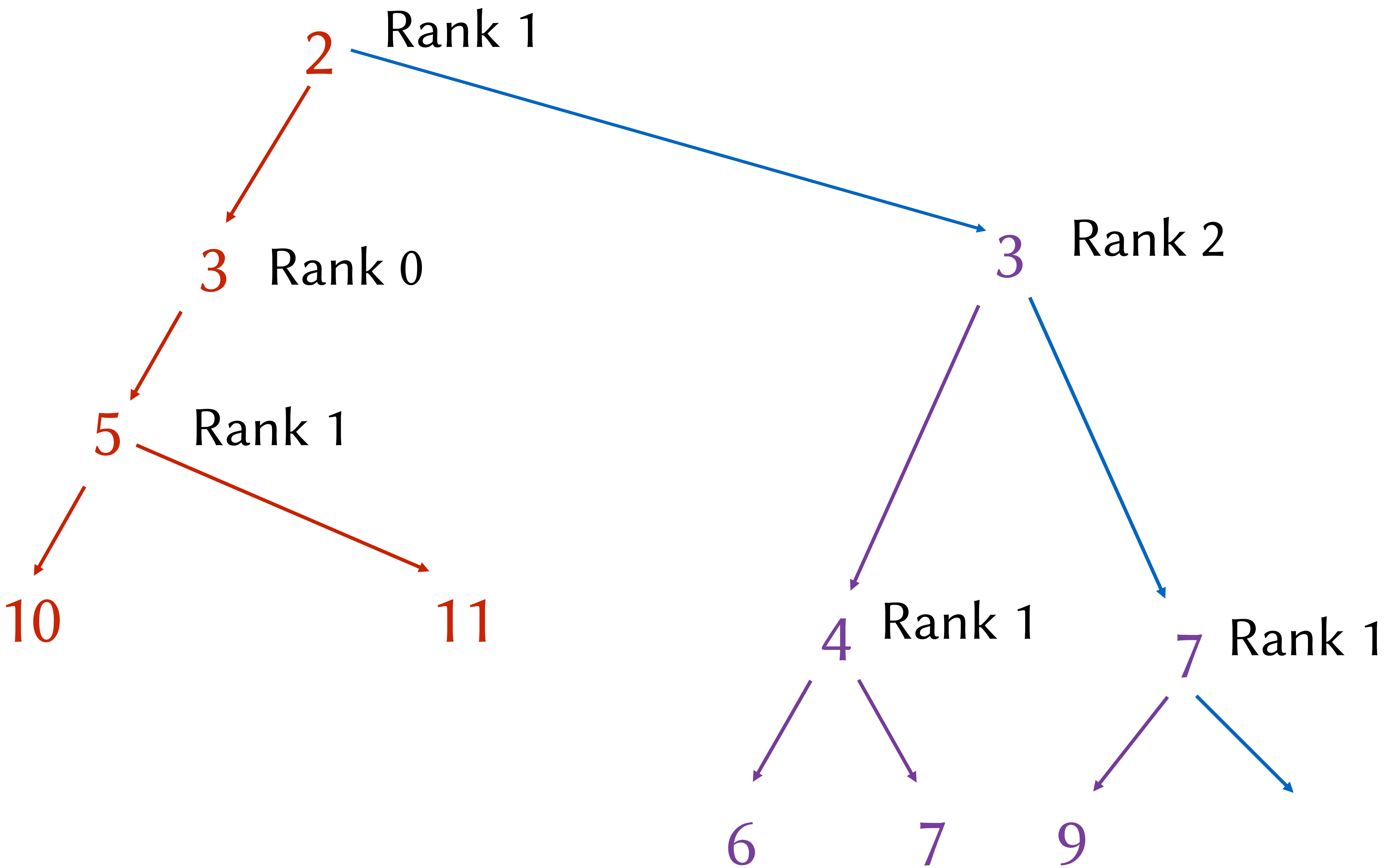
We want to merge these two trees.



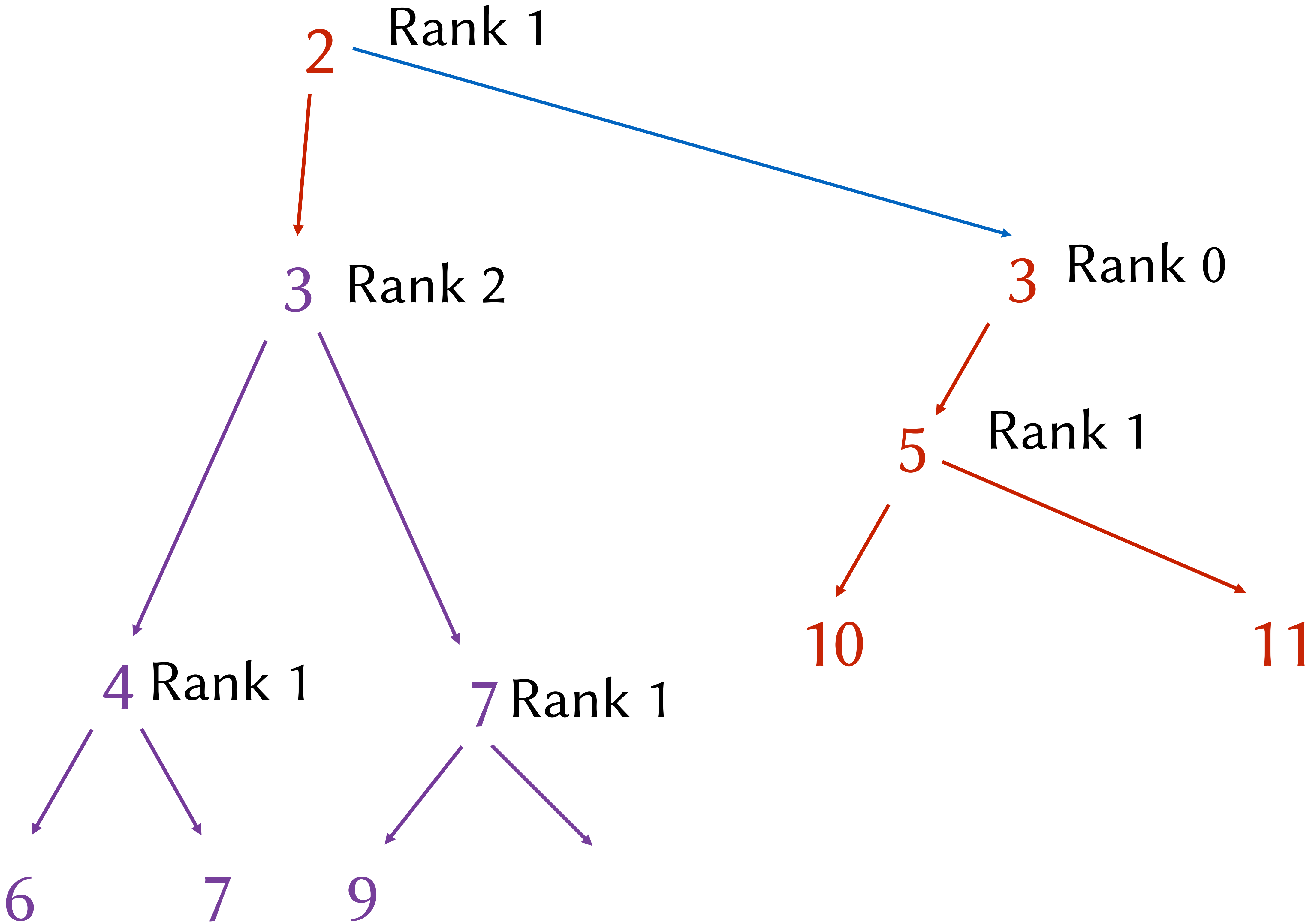
Which tree has the minimal root?



Merge below minimal root to maintain heap invariant.



What are our new node ranks?



Swap sub-trees to maintain rank invariant.

Leftist Heaps

```
case class Leaf[T <: Ordered[T]]() extends Heap[T] {
  def rank = 0
  def isEmpty = true

  def merge(that: Heap[T]) = that

  def min = throw new Error(
    "Attempt to call min on an empty heap")

  def deleteMin = throw new Error(
    "Attempt to call deleteMin on an empty heap")
}
```

Leftist Heaps

```
case class Branch[T <: Ordered[T]](
    rank: Int, x: T, left: Heap[T],
    right: Heap[T]) extends Heap[T] {

  def isEmpty = false

  def merge(that: Heap[T]) = {
    that match {
      case Leaf() => this
      case Branch(_, y, l, r) =>
        if (x <= y) makeBranch(x, left, right merge that)
        else makeBranch(y, l, this merge r)
    }
  }
  def min = x
  def deleteMin = left merge right
}
```

Leftist Heaps

```
abstract class Heap[T <: Ordered[T]] {
  def empty = Leaf[T]
  def isEmpty: Boolean

  def insert(element: T): Heap[T] =
    this merge Branch(1, element, empty, empty)

  def merge(that: Heap[T]): Heap[T]

  /* require (! isEmpty) */
  def min: T

  /* require (! isEmpty) */
  def deleteMin: Heap[T]

  def rank: Int

  def makeBranch(x: T, a: Heap[T], b: Heap[T]) = {
    if (a.rank >= b.rank) Branch(b.rank + 1, x, a, b)
    else Branch(a.rank + 1, x, b, a)
  }
}
```

Red-Black Trees

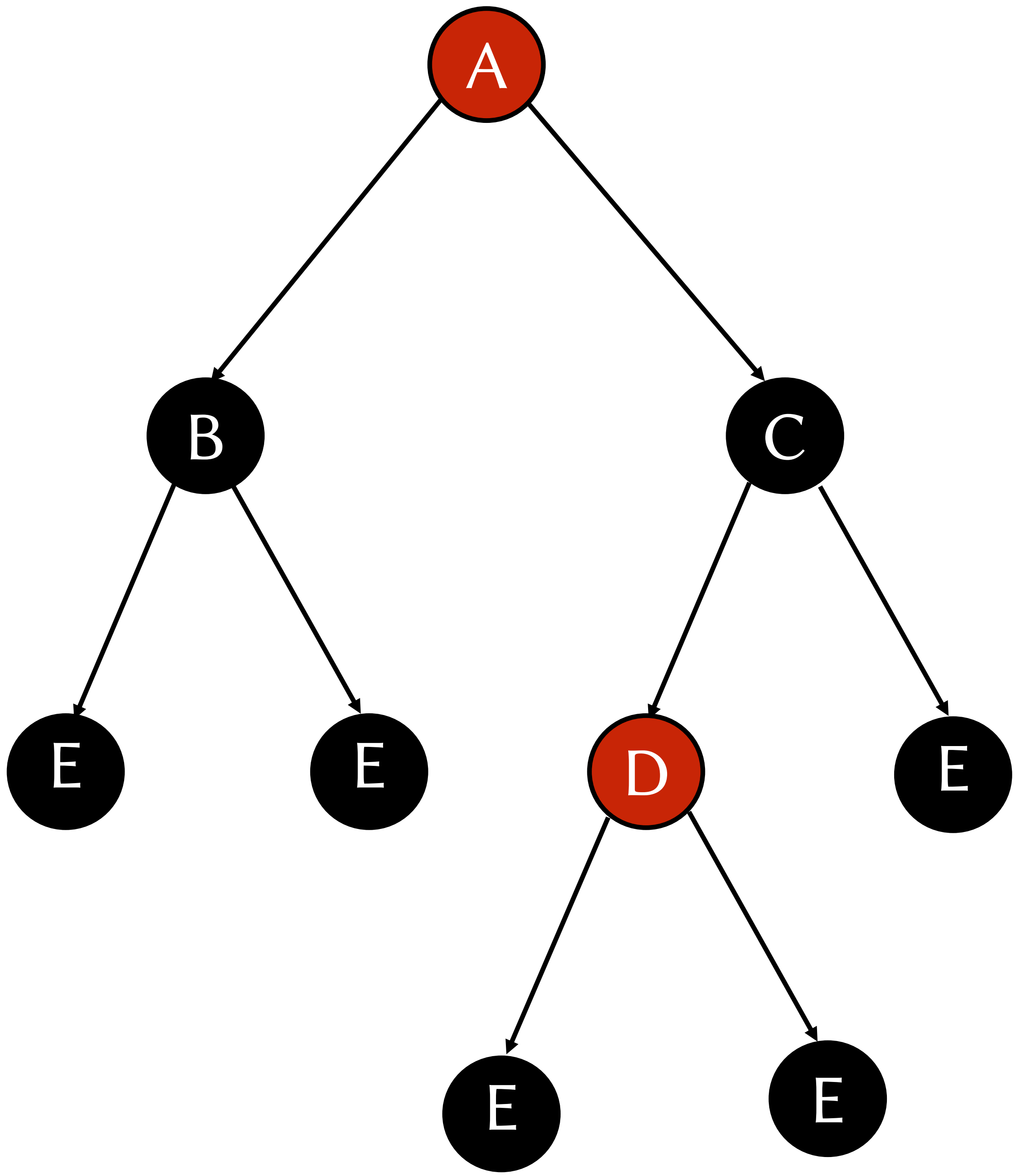
Red-Black Trees

- With naïve binary search trees, lookup can take O(n) time in the worst case
- We can fix this problem by rebalancing the trees as we add elements
- Red-Black trees are one approach to keeping the trees approximately balanced

Red-Black Trees

- Every node is colored either red or black
- All leaf nodes are black
- No red node has a red child
- Every path from the root to a leaf contains the same number of black nodes

An Example Red-Black Tree



Red-Black Trees

- These invariants imply that:
  - The longest possible path from the root to a leaf consists of an alternating sequence of red nodes and black nodes
  - The shortest possible path from the root to a leaf consists of all black nodes
- Thus, there is at most a factor of two difference in length between the shortest and longest paths

Red-Black Trees

```
sealed abstract class Color
case object Red extends Color
case object Black extends Color
```

Red-Black Trees

```
sealed abstract class Color
case object Red extends Color
case object Black extends Color
```

All subclasses of a sealed class must be defined in the same file as the sealed class.

Red-Black Trees

```
sealed abstract class Color
case object Red extends Color
case object Black extends Color
```

Pattern matching against a sealed class is checked to ensure exhaustiveness.

Strategy for Insertion

- We insert new elements as usual, but then rebalance the tree to maintain the red-black invariants
- At the end of the rebalancing, we recolor the root to black
  - This last step cannot violate our invariants

Red-Black Trees

```
abstract class Tree[T <: Ordered[T]] {
  def empty = Leaf[T]
  def contains(x: T): Boolean
  def insert(x: T): Tree[T] = insertChildren(x) match {
    case Branch(c,l,e,r) => Branch(Black, l, e, r)
  }
  def insertChildren(x: T): Branch[T]
}
```

We call a helper function insertChildren,
which performs the insertion and rebalancing.

Red-Black Trees

```
abstract class Tree[T <: Ordered[T]] {
  def empty = Leaf[T]
  def contains(x: T): Boolean
  def insert(x: T): Tree[T] = insertChildren(x) match {
    case Branch(c,l,e,r) => Branch(Black, l, e, r)
  }
  def insertChildren(x: T): Branch[T]
}
```

We take the result from insertChildren, ignore the color of the root and return a tree that is nearly identical except that the root is colored black.

Red-Black Trees

```
case class Leaf[T <: Ordered[T]]() extends Tree[T] {
  def contains(x: T) = false
  def insertChildren(x: T) = Branch(Red, this, x, this)
}
```

Red-Black Trees

```
case class Branch[T <: Ordered[T]]
(color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {

  def contains(x: T) = {
    if (x < element) left contains x
    else if (x > element) right contains x
    else true // x == element
  }
  …
}
```

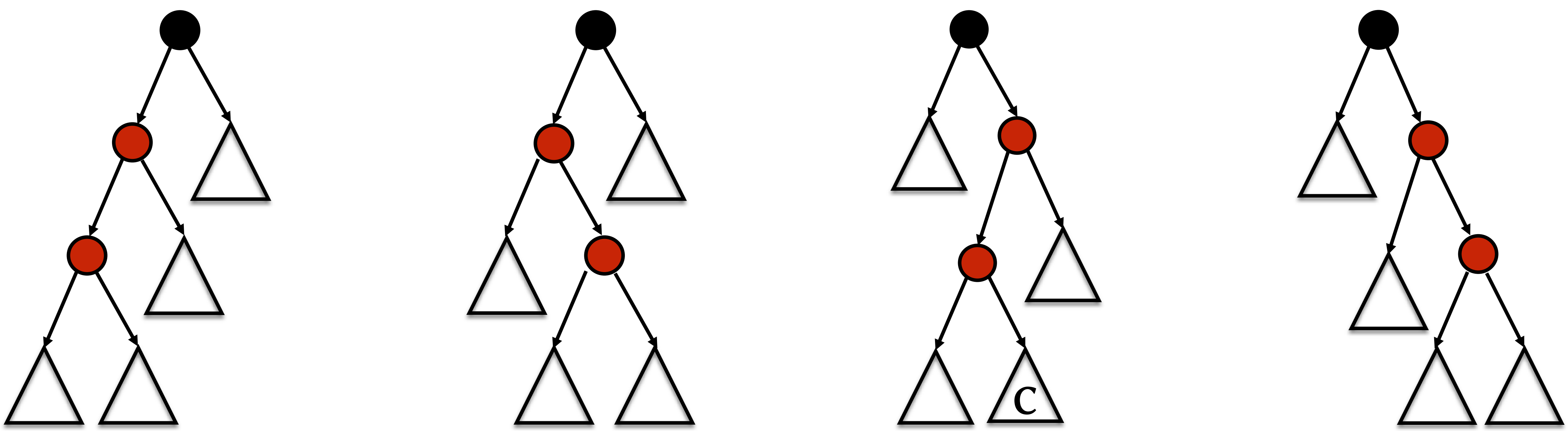
Red-Black Trees

```
case class Branch[T <: Ordered[T]]
(color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {
  …
  def insertChildren(x: T) = {
    if (x < element)
      balance(color, left insertChildren x, element, right)
    else if (x > element)
      balance(color, left, element, right insertChildren x)
    else this
  }
  …
}
```

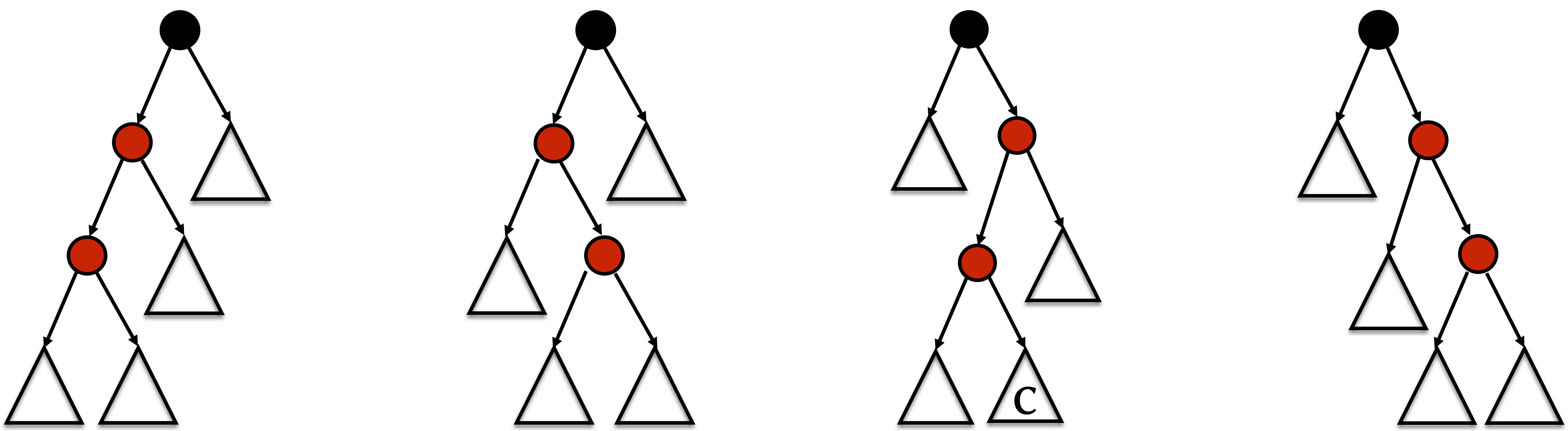
Rebalancing

- Because the base case of insertChildren (at a leaf node) always inserts a red node, the number of black nodes along each path is unaffected
- However, the new tree might contain a red node with a red child

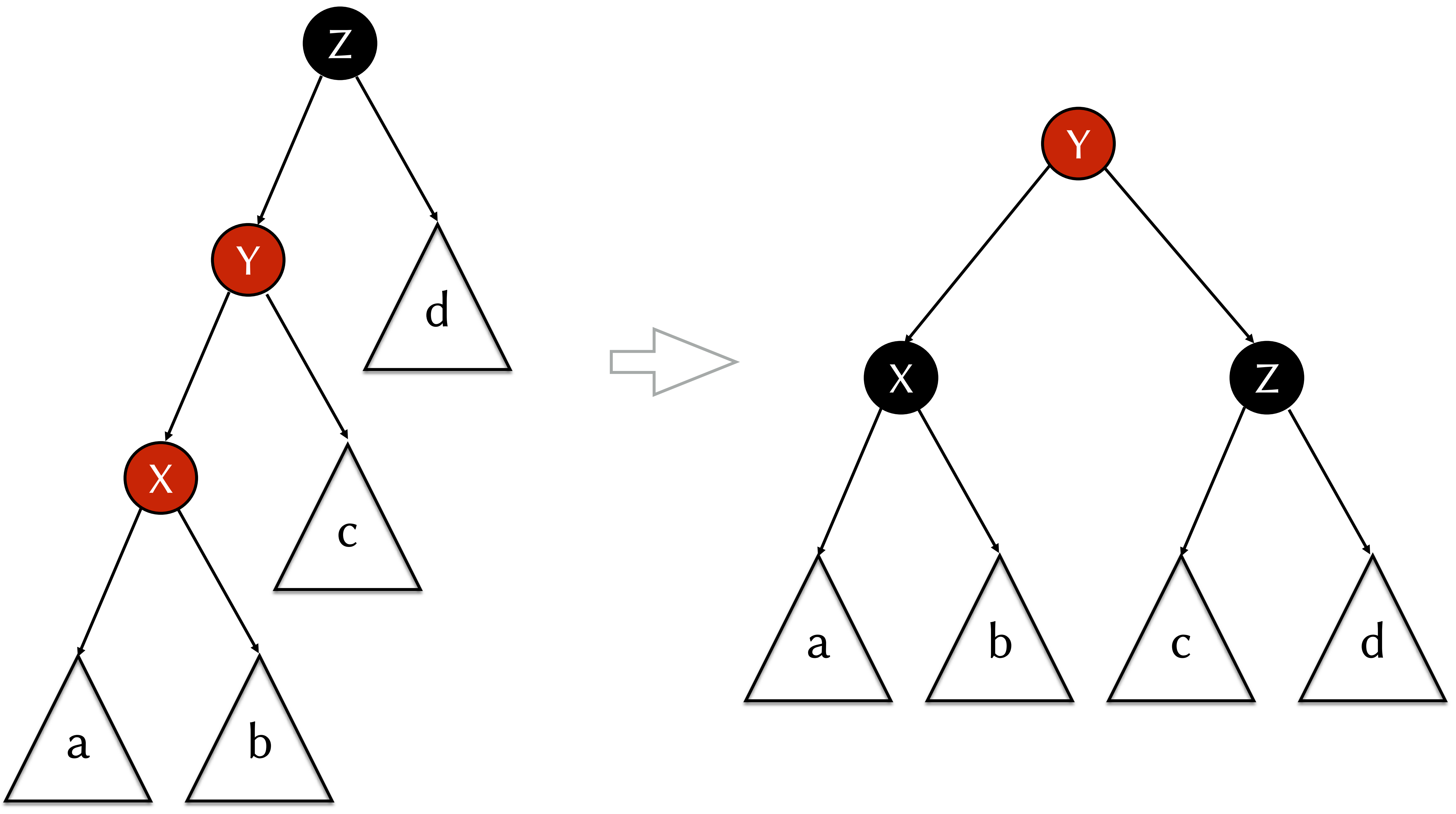
Rebalancing:
There are Four Cases to Consider



Rebalancing:
There are Four Cases to Consider



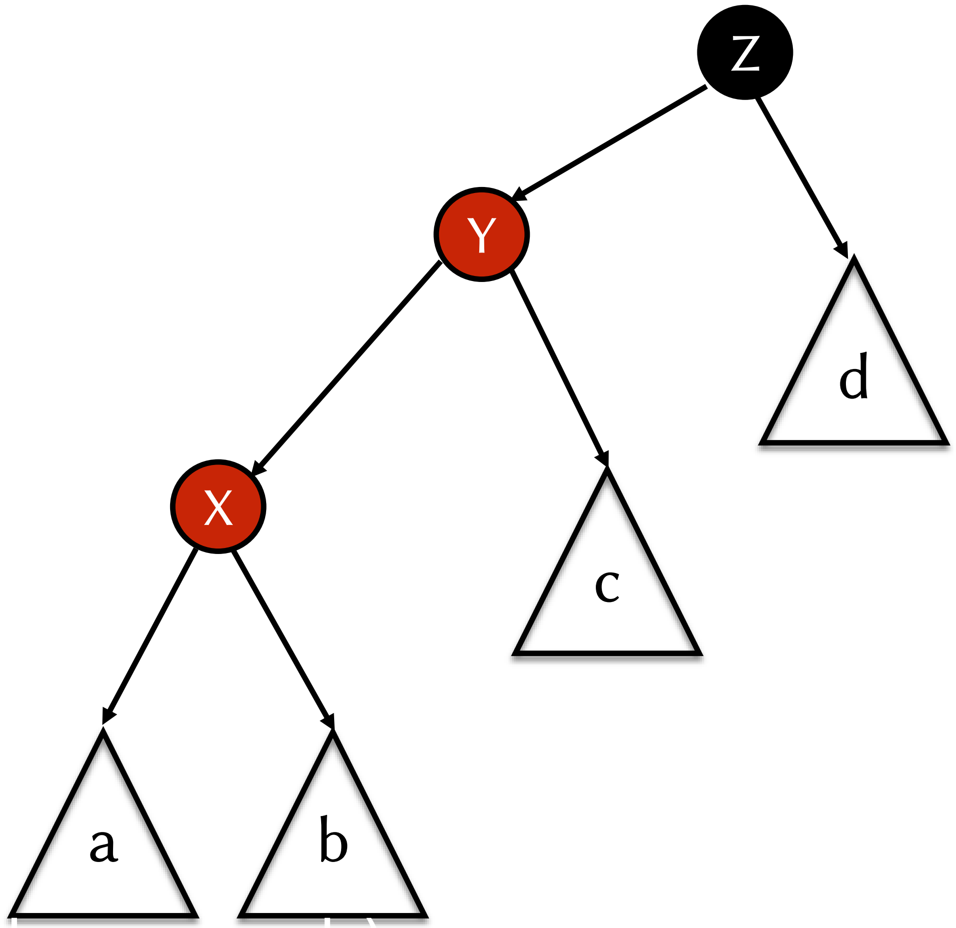
We use pattern matching to enumerate the cases.



Z
Y
X
d
c
a
b
Y
X
Z
a
b
c
d

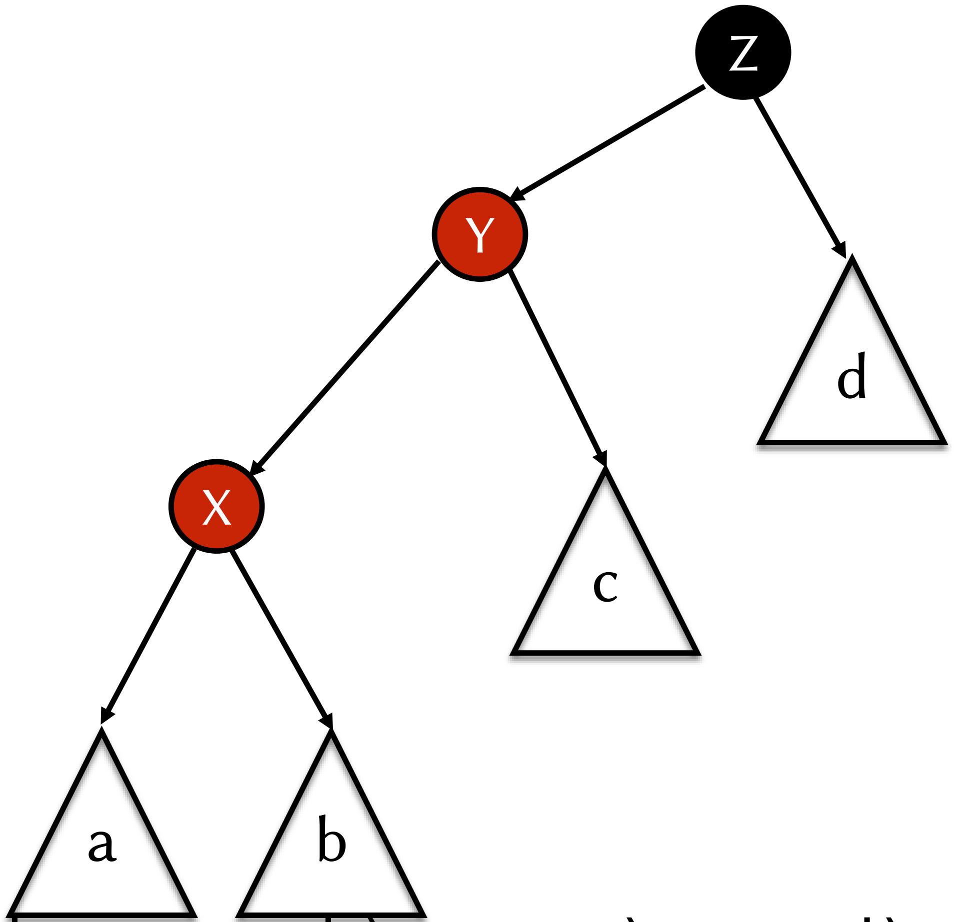
```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {












          …
    }
  }
  …
}
```

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```

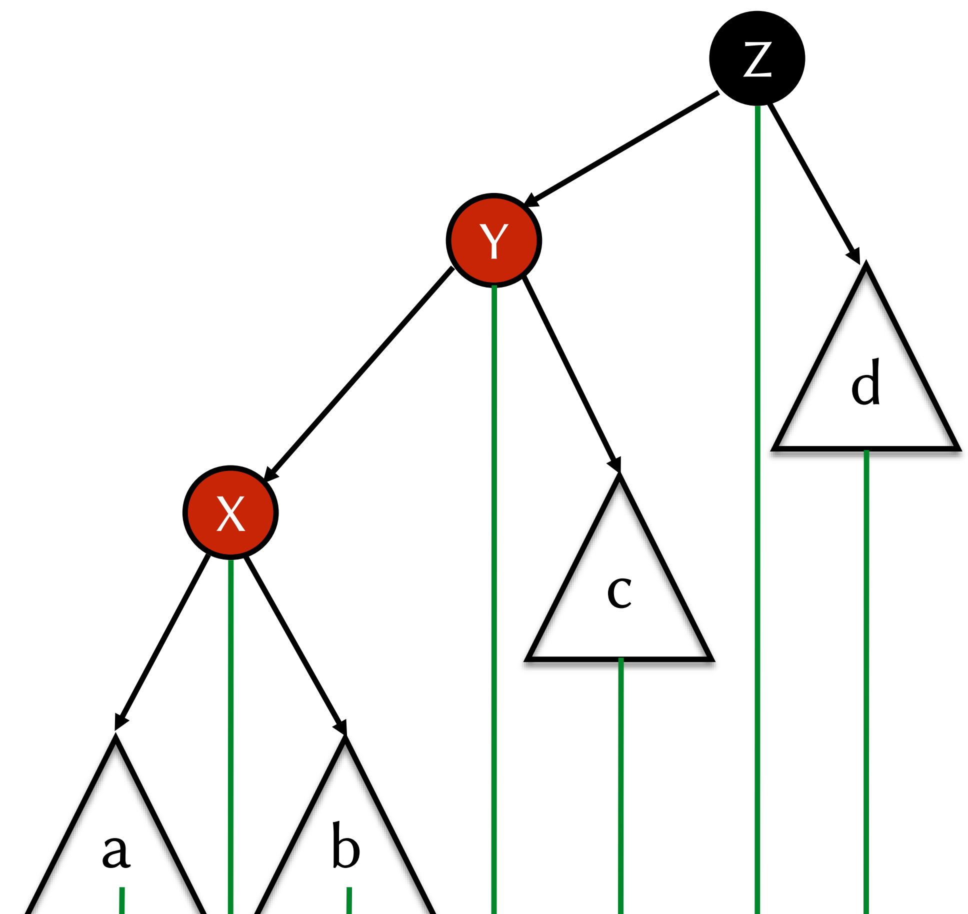



```
        …
    }
  }
  …
}
```

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>

      …
    }
  }
  …
}
```

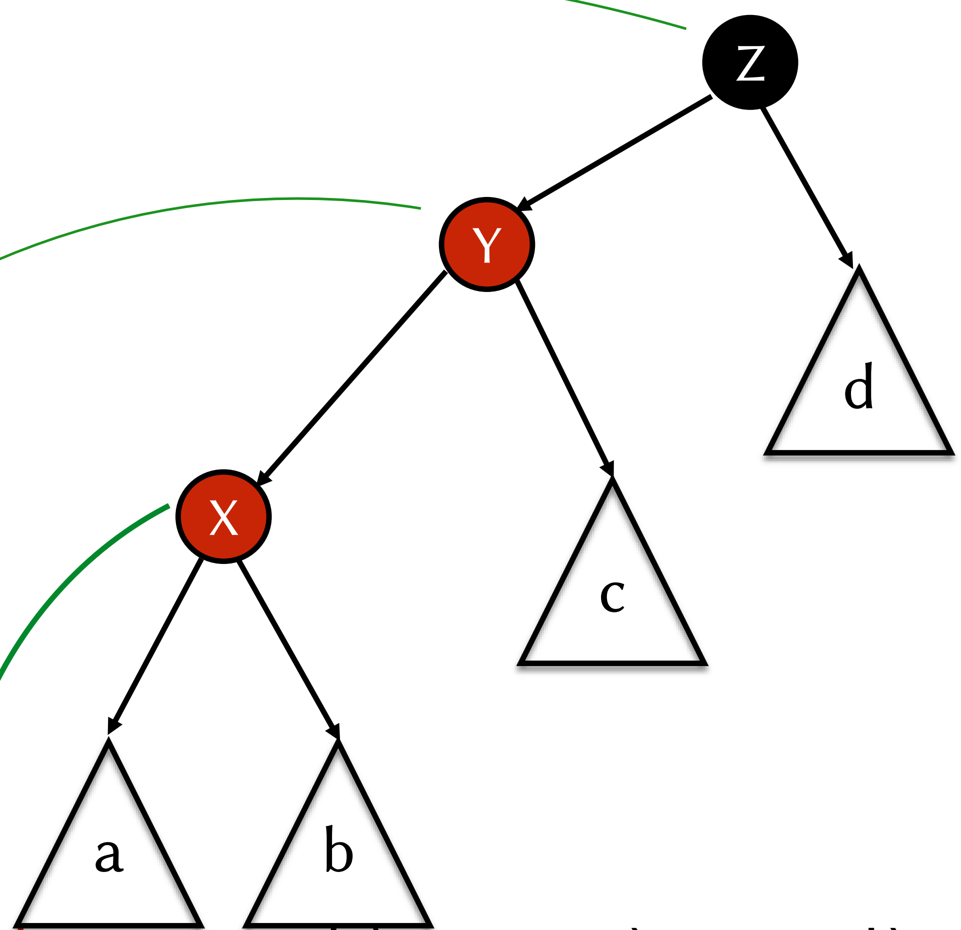


```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```



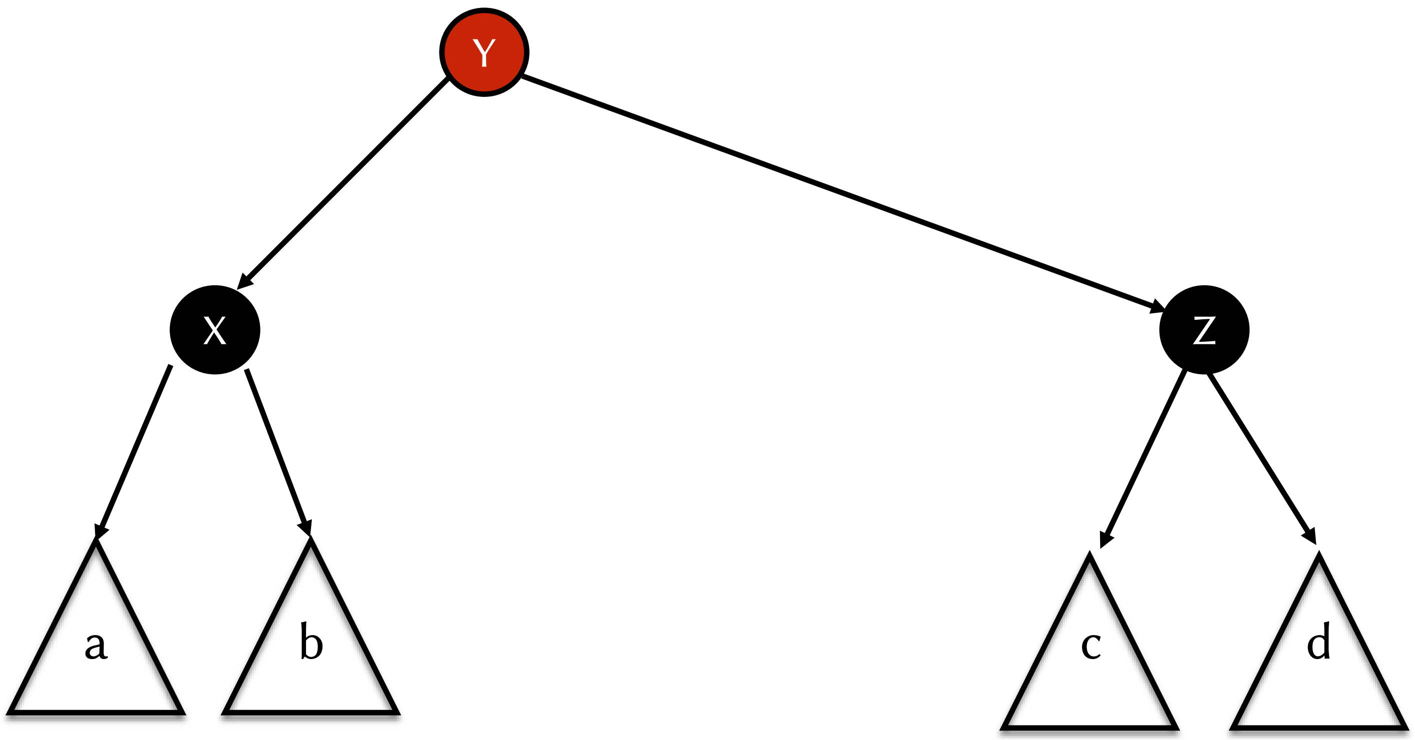

```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>

      …
    }
  }
  …
}
```

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```



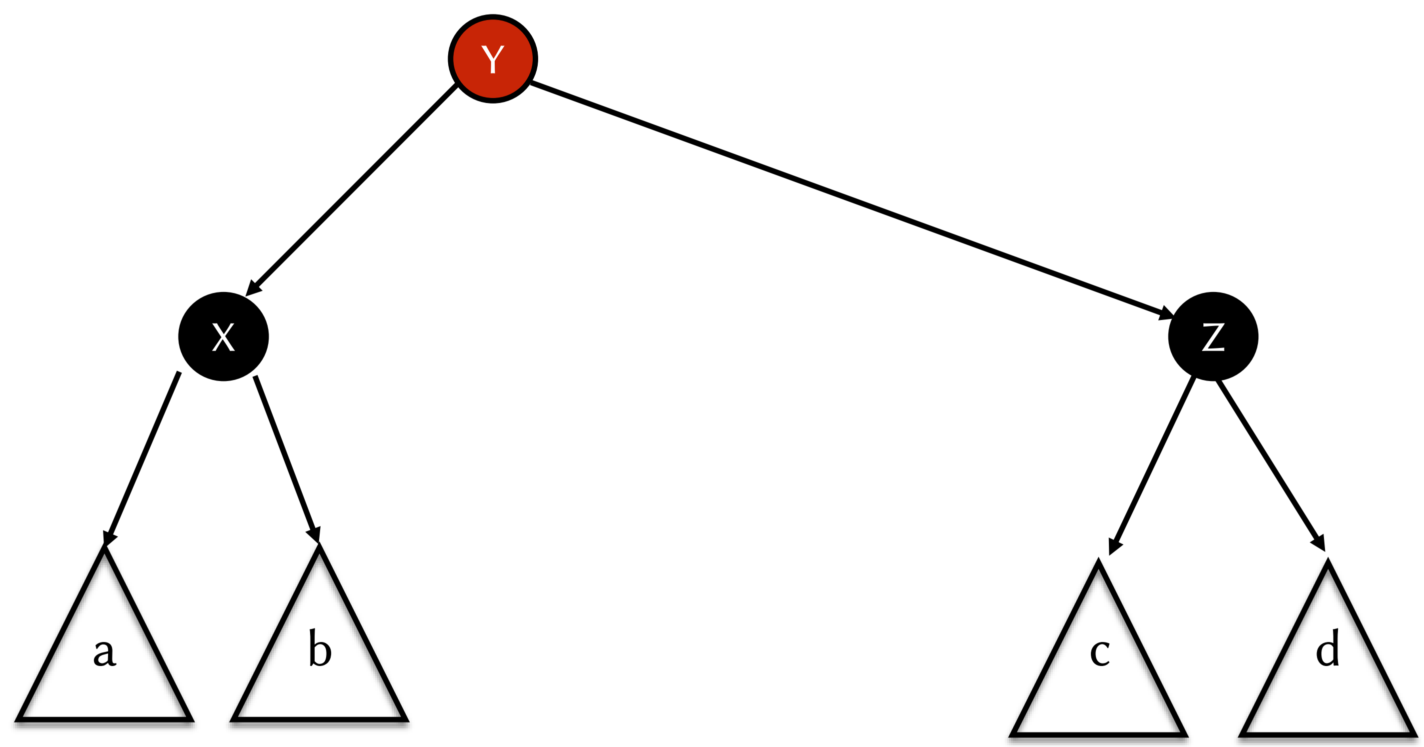

```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>

      …
    }
  }
  …
}
```

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```



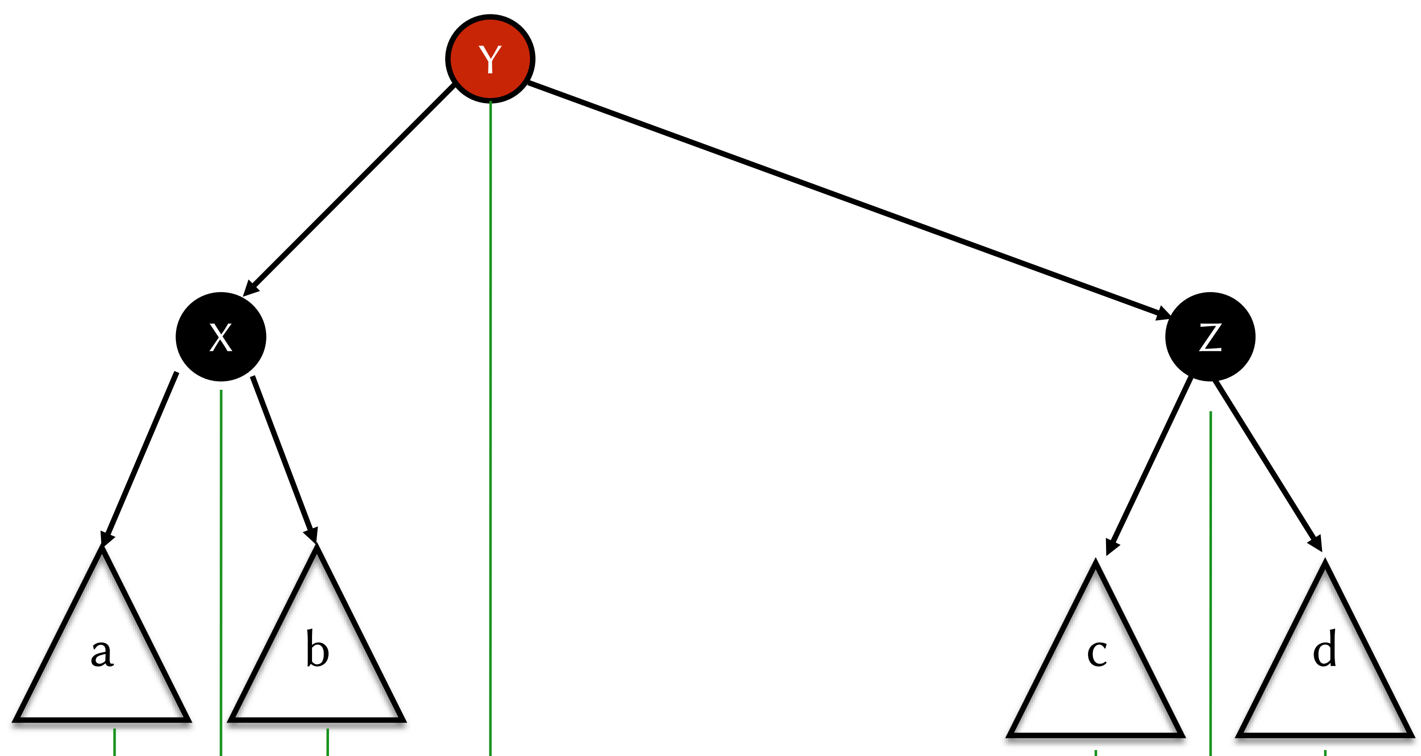

```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>

      …
    }
  }
  …
}
```

```scala
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```



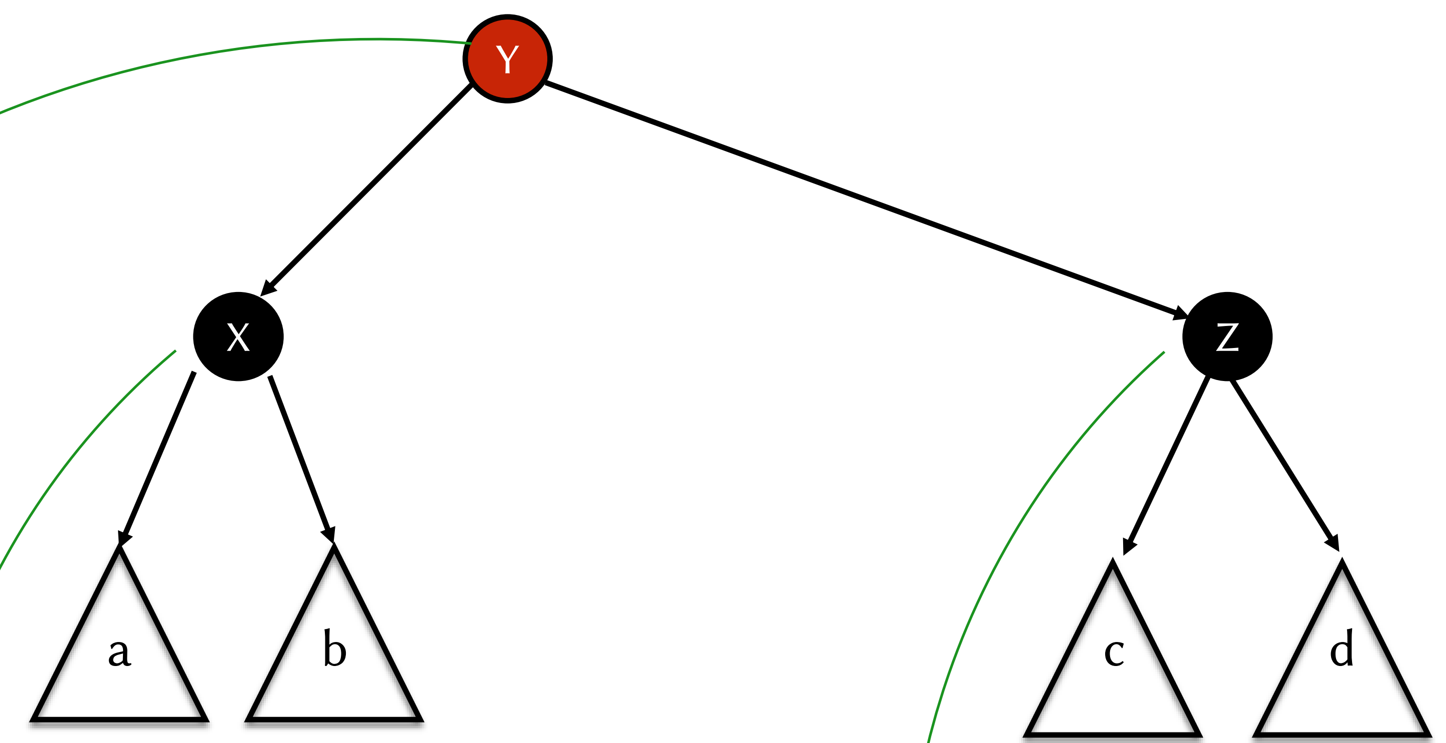

```scala
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
  }
  …
}
```

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```

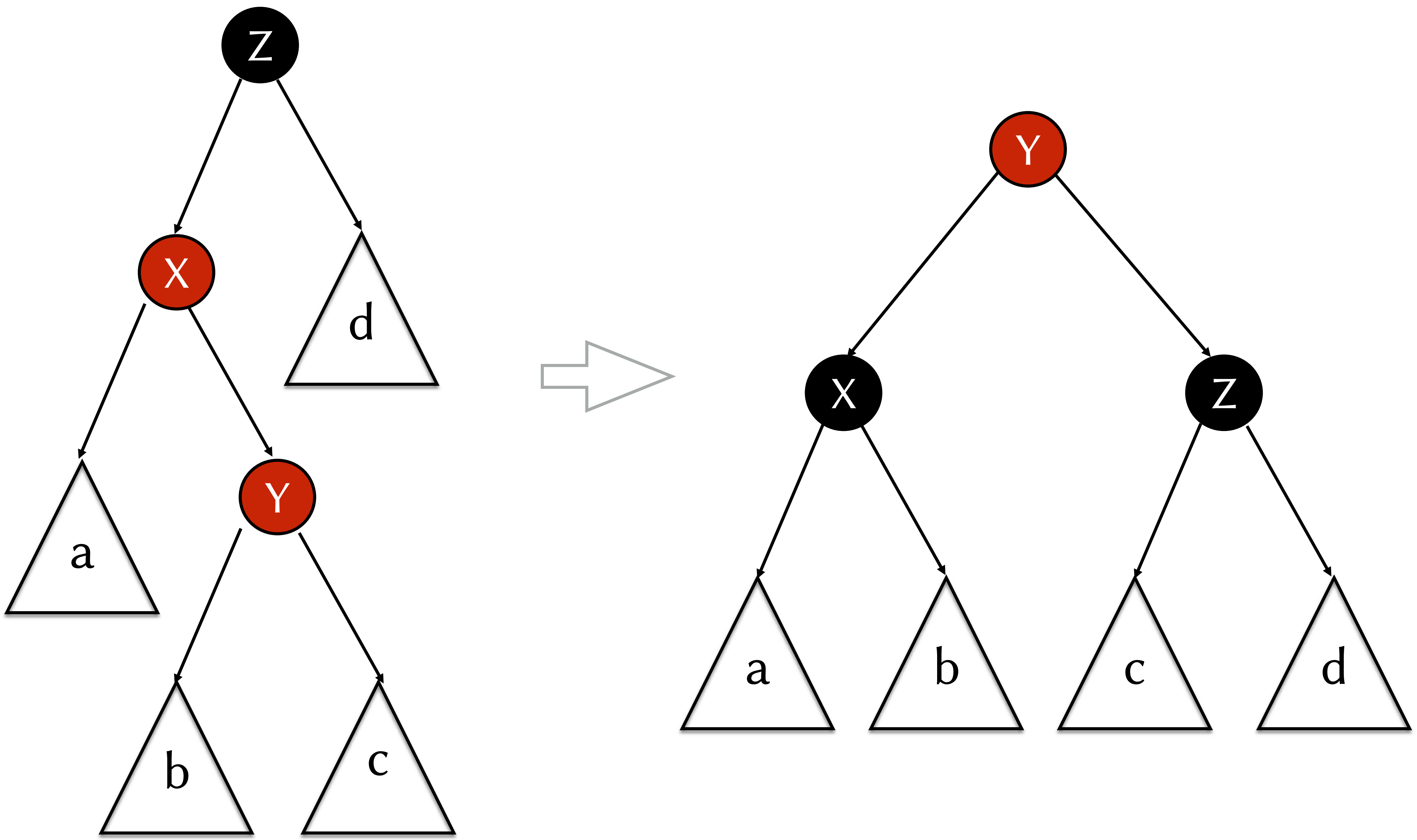



```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
  }
  …
}
```

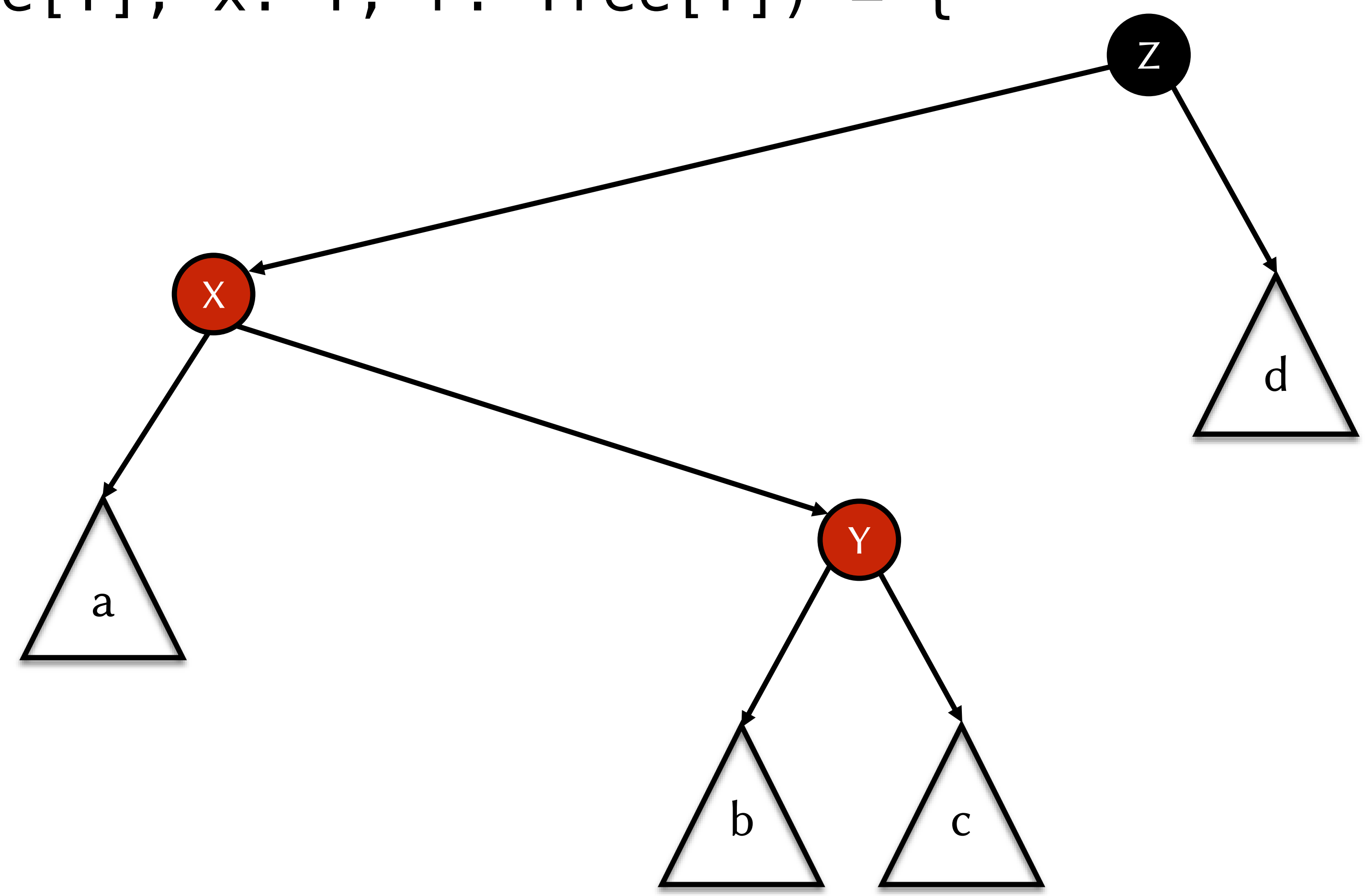
```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```




```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
  }
  …
}
```

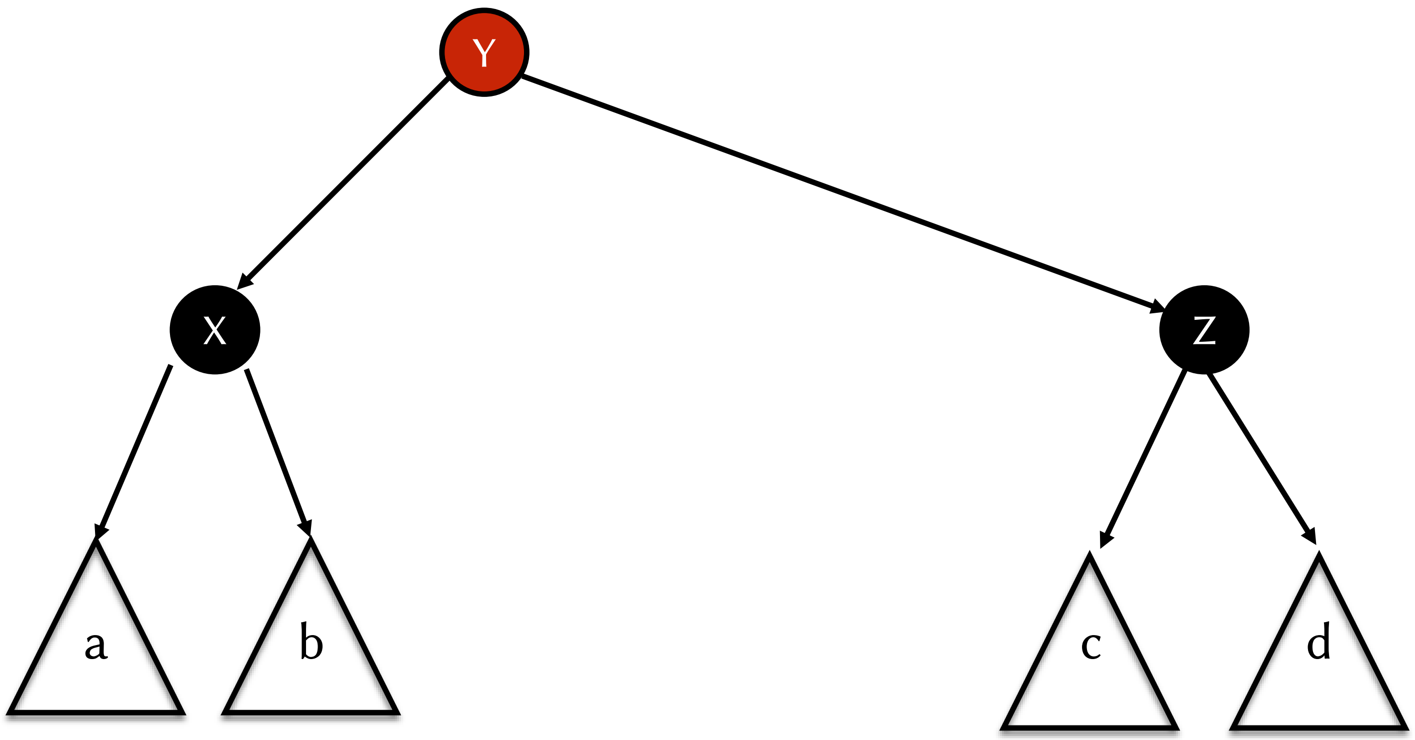

Z
X
d
a
Y
b
c
Y
X
Z
a
b
c
d

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```

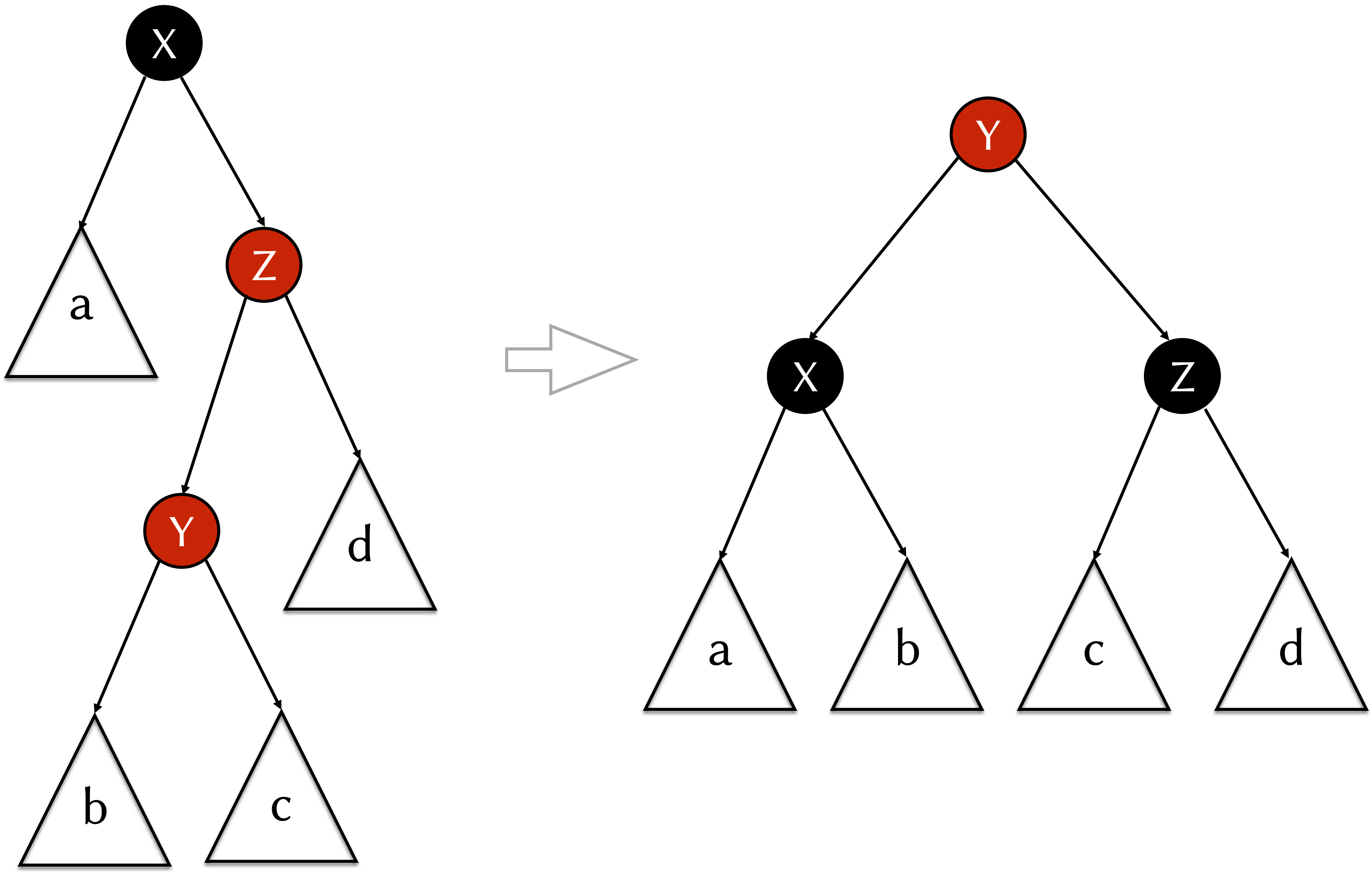



```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>


      …
    }
  }
  …
```

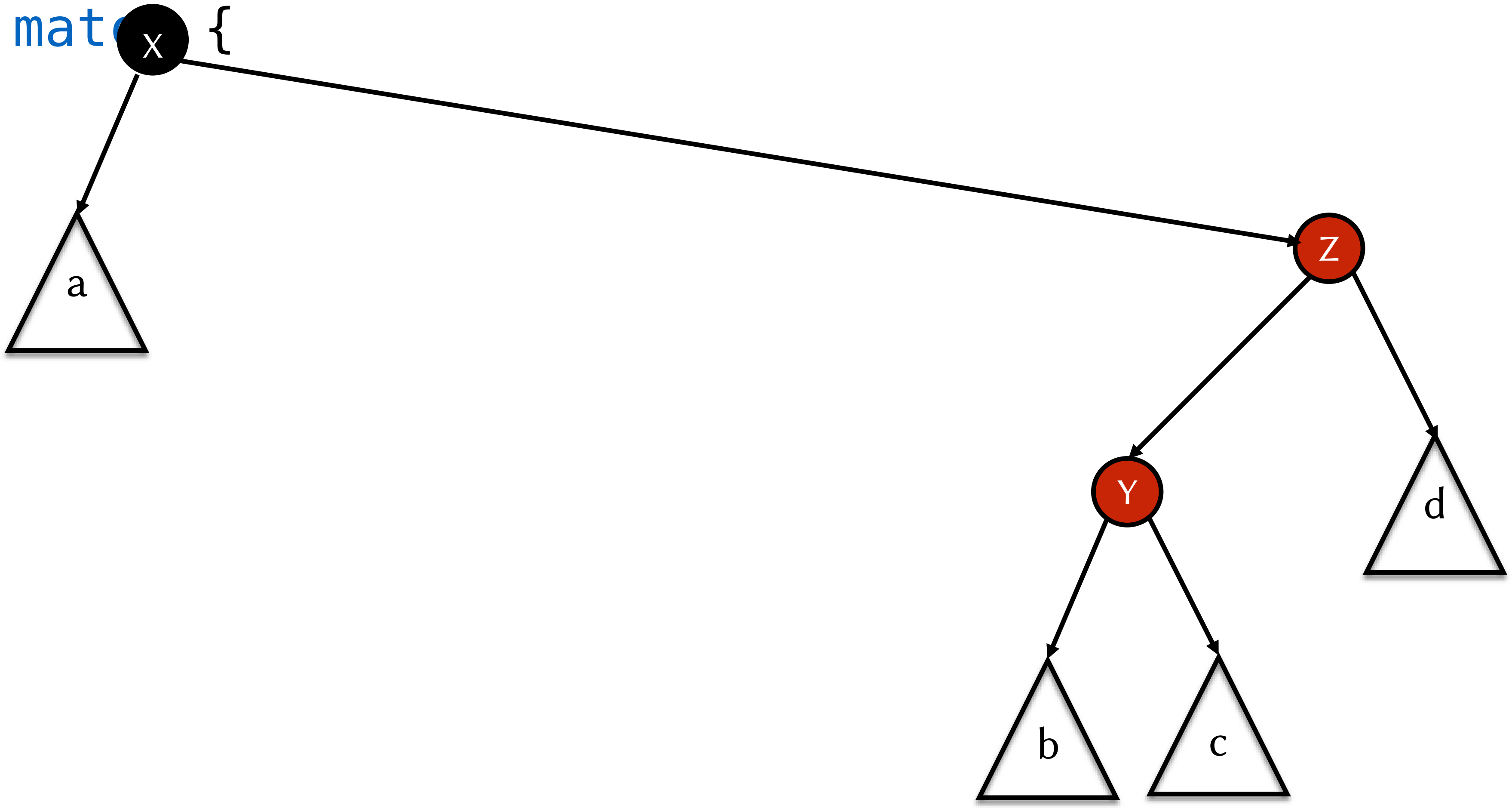
```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```




```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
  }
  …
}
```

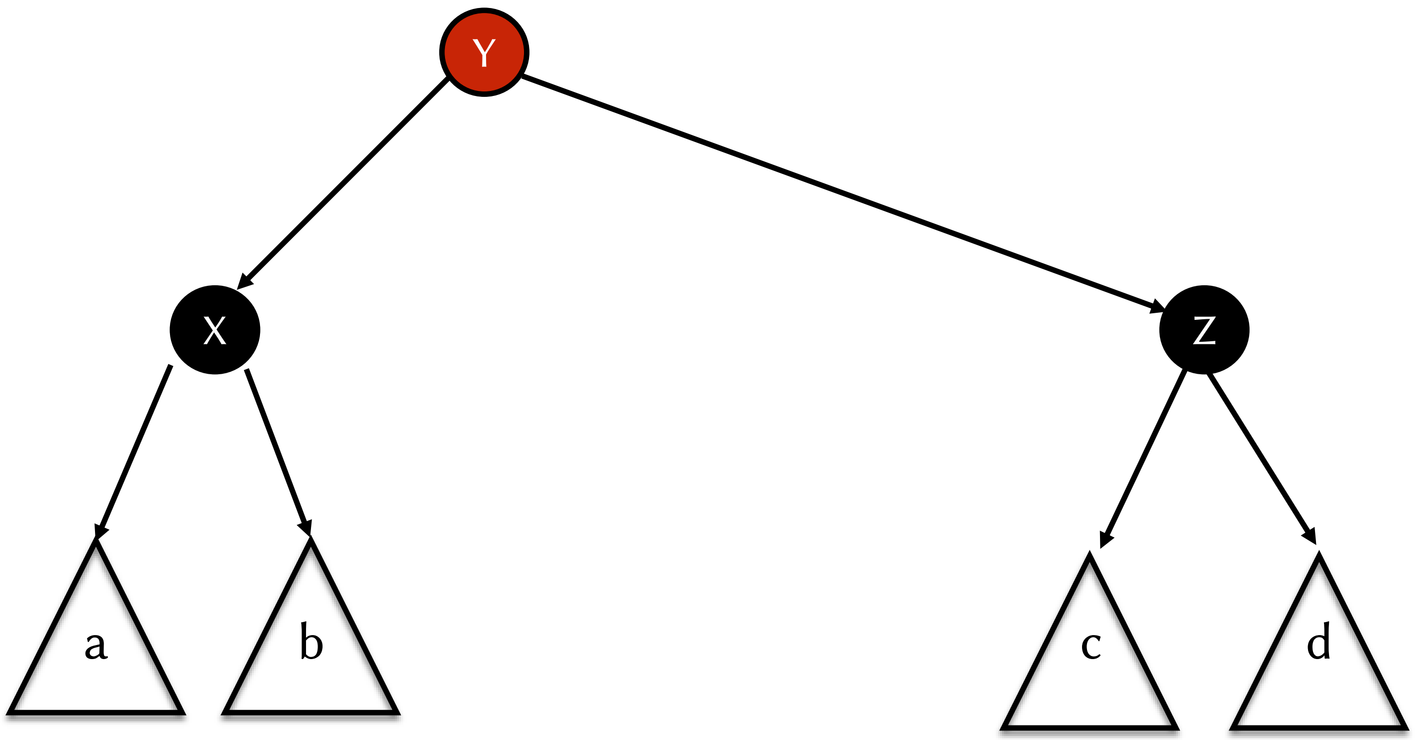

X
a
Z
Y
d
b
c
Y
X
Z
a
b
c
d

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```

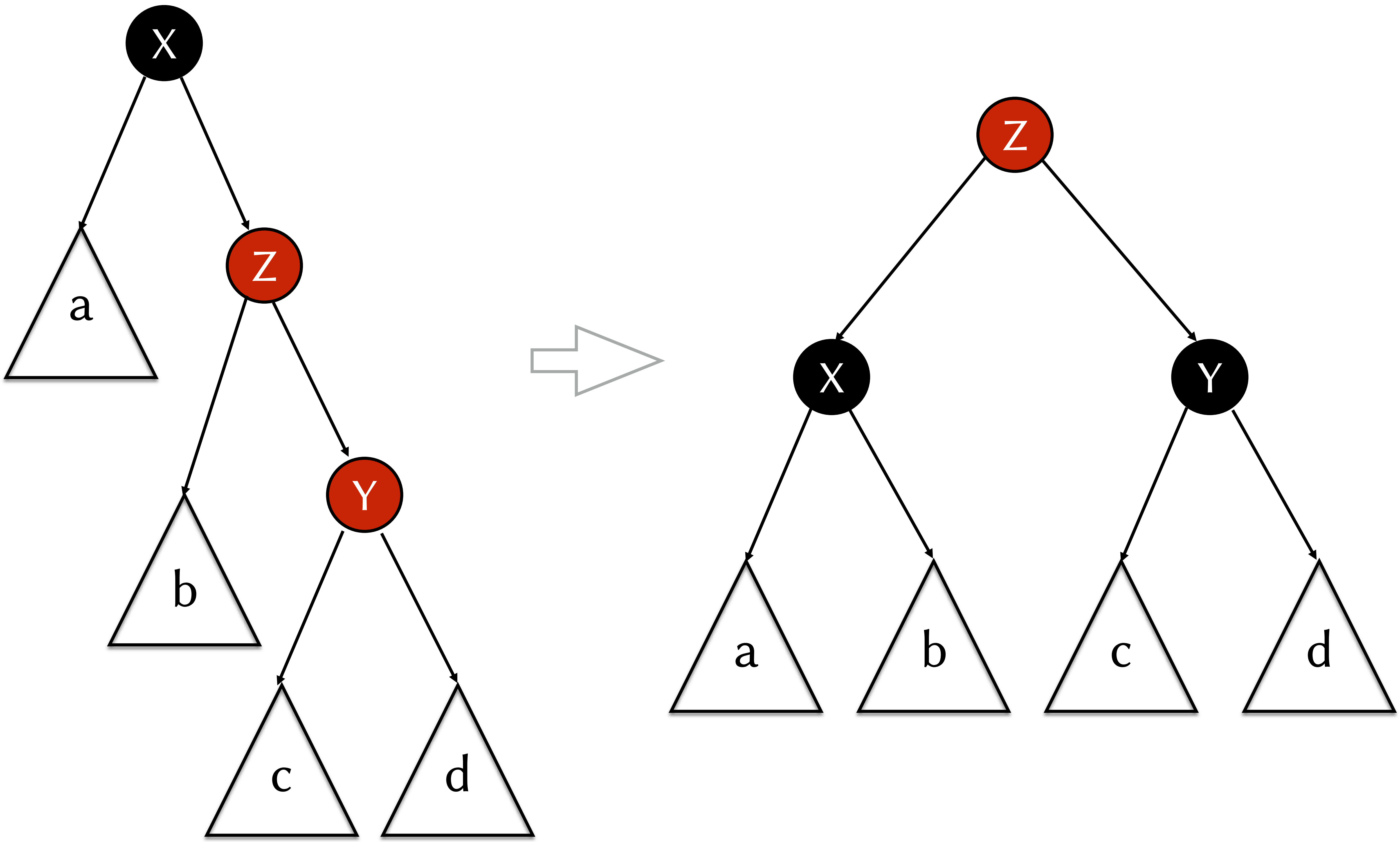



```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>


      …
    }
  }
```

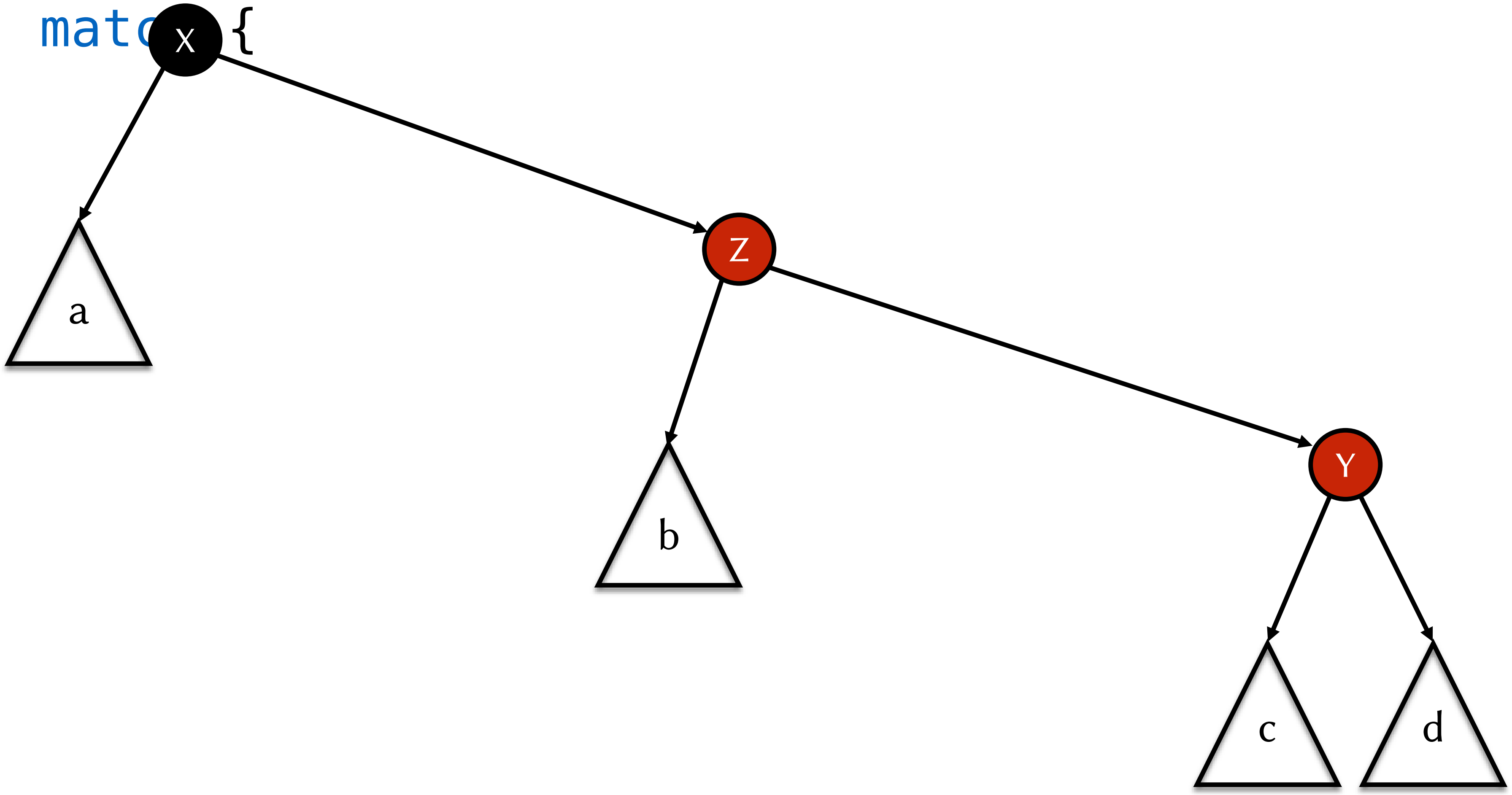
```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```




```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
  }
```

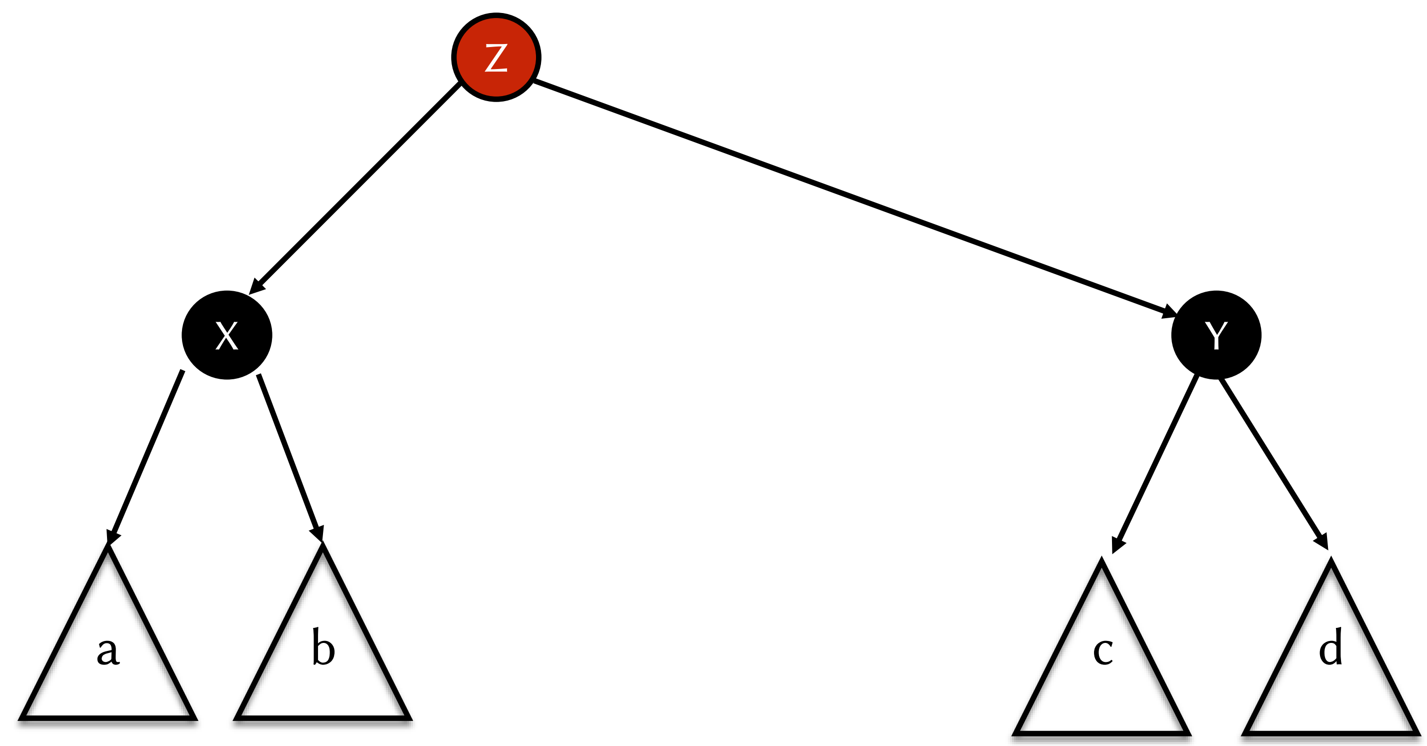

X
a
Z
b
Y
c
d
Z
X
Y
a
b
c
d

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```




```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>

      …
    }
}
```

```
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
```




```
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
```

```scala
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      …
    }
  }
```

```scala
def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case _ => Branch(c, l, x, r)
    }
}
```

Red-Black Trees

```
case class Branch[T <: Ordered[T]]
(color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {
  …
  def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case _ => Branch(c, l, x, r)
    }
  }
  …
}
```

Unfortunately, all four consequences are syntactically identical

Red-Black Trees

```
case class Branch[T <: Ordered[T]]
(color: Color, left: Tree[T], element: T, right: Tree[T])
extends Tree[T] {
  …
  def balance(c: Color, l: Tree[T], x: T, r: Tree[T]) = {
    (c, l, x, r) match {
      case (Black, Branch(Red, Branch(Red, a, x, b), y, c), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, Branch(Red, a, x, Branch(Red, b, y, c)), z, d) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, Branch(Red, b, y, c), z, d)) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case (Black, a, x, Branch(Red, b, y, Branch(Red, c, z, d))) =>
        Branch(Red, Branch(Black, a, x, b), y, Branch(Black, c, z, d))
      case _ => Branch(c, l, x, r)
    }
  }
  …
}
```

In some languages (such as ML) we could factor this out with "or" patterns

Discussion

- This implementation of red-black trees is dramatically simpler than most imperative approaches:
  - Imperative approaches typically include eight cases, branching on the color of the red parent's sibling
  - These cases help to avoid some assignment and copying in an imperative setting